A Cross-Layer Framework for Sensor Networks

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Andreas Jürgen Lachenmann

aus Nürtingen

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter:	Prof. Dr. rer. nat. habil. Pedro José Marrón
Tag der mündlichen Prüfung:	24. April 2008

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2008

Acknowledgments

First of all, I would like to thank my advisor, Kurt Rothermel, for giving me the opportunity to work on this dissertation in his group. He provided the right combination of valuable feedback and of the freedom to pursue own ideas. Special thanks also go to Pedro José Marrón for being a great co-advisor.

I would like to thank my colleagues in the Distributed Systems Department, especially Matthias Gauger, Klaus Herrmann, Daniel Minder, Olga Saukh, and Robert Sauter. They created a unique environment where working was fun. In addition, I am grateful for their discussions and comments that helped to greatly improve my work. I also want to thank the students who contributed to parts of this thesis.

Furthermore, I would like to thank Landesstiftung Baden-Württemberg for partially funding my work.

Finally, I thank my parents and my brother for their continuing support throughout my studies and the work on this dissertation.

Contents

List of Figures

List of Tables

Abstract

Cross-layer interactions are often used in wireless sensor networks. They help to optimize energy consumption, deal with memory limitations, and consider the special properties of wireless communication. However, cross-layer interactions have the disadvantage of negatively affecting desirable properties of the software design like modularity and reusability. In the extreme, applications consist of a monolithic piece of code that is hard to develop and impossible to maintain.

Therefore, this thesis investigates different approaches to address the negative side-effects of cross-layer interactions. In particular, it develops a framework that pursues three different strategies.

First, it tries to preserve modularity and increase reusability by decoupling components that exchange data. This strategy is realized by *TinyXXL*, a programming abstraction for cross-layer data exchange. This part of the framework has been created based on an analysis of cross-layer interactions in existing applications. With some compile-time optimizations *TinyXXL* can reduce both energy and memory consumption compared to an application built from reusable components. Using *Neidas*, a novel neighborhood data sharing algorithm, it offers a comprehensive system for data exchange among the layers of a single node and with neighboring nodes.

Second, the framework relaxes one of the constraints that often lead to cross-layer interactions and, thus, reduces the need to apply them. Specifically, it includes *ViMem*, a flash-based virtual memory system that helps to reduce memory limitations and tries to optimize the memory layout.

Finally, the third strategy is to partially move energy concerns into the system software. For this purpose the framework includes *Levels*, an abstraction to specify optional functionality which allows to accurately meet a user-defined lifetime goal. If necessary, *Levels* deactivates functionality in order to reach that target lifetime. Furthermore, it includes a distributed algorithm that helps to provide a constant application quality over the total network lifetime.

Deutsche Zusammenfassung

1. Einleitung

In den letzten Jahren wurden drahtlose Sensornetze entwickelt, die dazu dienen, Daten aus der realen Welt zu sammeln. Diese Netze bestehen aus so genannten Sensorknoten, kleinen batteriebetriebenen Rechnern, die über Funk miteinander kommunizieren. Vielversprechende Anwendungen für Sensornetze sind z. B. die Beobachtung von Naturereignissen [Werner-Allen et al. 2006a] und von Tieren in ihrem natürlichen Lebensraum [Szewczyk et al. 2004; Juang et al. 2002] sowie die Strukturüberwachung von Brücken und Gebäuden [Marrón et al. 2005c; Xu et al. 2004; Kim et al. 2007] und die Überwachung von Maschinen und Industrieanlagen [Krishnamurthy et al. 2005].

Ein typischer Sensorknoten wie z. B. ein Mica2-Knoten besteht aus einem Mikrocontroller mit einer Taktfrequenz von wenigen Megahertz und wenigen Kilobyte Arbeitsspeicher, einem nichtflüchtigen Flashspeicher, einer Funkschnittstelle und anwendungsabhängigen Sensoren. Da viel Anwendungen für Sensornetze über einen langen Zeitraum ohne menschlichen Eingriff betrieben werden sollen und weil die Energieversorgung meist über Batterien erfolgt, ist Energie eine stark beschränkte Ressource. Dies führt auch dazu, dass beispielsweise energiesparende Mikrocontroller mit nur wenig Arbeitsspeicher eingesetzt werden.

Diese Ressourcenbeschränkungen erfordern, dass die Software der Sensorknoten optimiert wird. In den Netzwerkprotokollen werden hierzu häufig schichtenübergreifende Optimierungen eingesetzt (so genannte Cross-Layer-Interaktionen). Im Gegensatz zu einer strikt getrennten Schichtenarchitektur arbeiten dabei die verschiedenen Schichten enger zusammen. So können sie zum Beispiel Daten gemeinsam benutzen, was sowohl Platz im Arbeitsspeicher als auch Energie für das Sammeln der Daten einzusparen hilft. Weiterhin erlauben Cross-Layer-Interaktionen, besser mit den Besonderheiten der drahtlosen Kommunikation umzugehen. Beispielsweise haben hier – im Gegensatz zur Annahme von TCP – Paketverluste häufig andere Ursachen als eine Überlastung der Verbindung.

Neben ihren allgemein anerkannten Vorteilen, die viele Anwendungen auf den ressourcenbeschränkten Sensorknoten überhaupt erst möglich machen, haben Cross-Layer-Interaktionen allerdings auch Nachteile. Insbesondere haben sie negative Auswirkungen auf wünschenswerte Eigenschaften der Softwarearchitektur [Kawadia and Kumar 2005]: die Modularität wird verringert und einzelne Schichten können nicht mehr ohne weiteres ausgetauscht werden. Im Extremfall entsteht eine monolithische Anwendung, die nicht mehr wartbar ist. Ziel dieser Arbeit ist es deshalb, Programmierabstraktionen und Systemsoftware zu entwickeln, die diese negativen Auswirkungen zu vermeiden helfen.

Zunächst werden verschiedene Sensornetzanwendungen analysiert, um die darin verwendeten Cross-Layer-Interaktionen zu identifizieren und zu klassifizieren. Dann wird ein Framework vorgestellt, das drei verschiedene Strategien anwendet, um die negativen Auswirkungen von Cross-Layer-Interaktionen zu verringern. Im Unterschied zu existierenden Ansätzen [Conti et al. 2004; Kumar et al. 2006; Su and Lim 2006] beschränkt sich das hier beschriebene Cross-Layer-Framework allerdings nicht darauf, diese negativen Auswirkungen zu lindern, sondern befasst sich mit ihren Ursachen und versucht häufige Gründe für solche Interaktionen aus dem Zuständigkeitsbereich des Anwendungsentwicklers zu entfernen. Die drei Strategien des Frameworks sehen dabei wie folgt aus:

Erstens wird versucht, Schichten und Softwarekomponenten, die Daten austauschen, zu entkoppeln. Damit bleibt trotz der Interaktion die Modularität größtenteils erhalten. Diese Art der Interaktion wurde ausgewählt, da sich in unserer Analyse von Anwendungen zeigte, dass es hierbei ein großes Potential für Optimierungen gibt und dass diese Interaktionsform noch nicht ausreichend von bestehenden Programmiersprachen unterstützt wird. Diese Strategie wird durch *TinyXXL* und das *TinyStateRepository* umgesetzt. Zusammen mit *Neidas*, einem Algorithmus für die Verteilung von Daten auf Nachbarknoten, bietet *TinyXXL* ein System an, das sowohl für den Datenaustausch innerhalb eines Knotens als auch in der Nachbarschaft geeignet ist. *Neidas* versucht die übertragene Datenmenge zu minimieren und nutzt dabei die Broadcast-Übermittlung in Funknetzen aus.

Die zweite Strategie des Frameworks ist es, eine der Randbedingungen, die häufig zu Cross-Layer-Interaktionen führt, zu lockern. Dabei wird die Speicherbeschränkung teilweise aufgehoben, indem mit *ViMem* ein flash-basiertes virtuelles Speichersystem entwickelt wird. Wenn die Speicherbeschränkungen weniger streng sind, werden weniger Cross-Layer-Interaktionen eingesetzt, und es treten auch keine negativen Nebenwirkungen auf. *ViMem* versucht mit Hilfe einer Heuristik, die Anordnung von Daten im Speicher so zu optimieren, dass möglichst wenige Seiten vom Flash-Speicher gelesen bzw. dorthin geschrieben werden müssen. Dabei werden die spezifischen Eigenschaften von Sensornetzen und Flash-Speichern besonders berücksichtigt.

Als dritte Strategie verfolgt das Framework das Ziel, manche Cross-Layer-Interaktionen unnötig zu machen, indem Energie-Aspekte – ein weiterer häufiger Grund für Cross-Layer-Interaktionen – in die Systemsoftware verschoben werden. Diese Strategie wird mit *Levels* umgesetzt, das die Möglichkeit bietet, ein Ziel für die Lebensdauer jedes Knotens vorzugeben. Das System versucht dann ohne Zutun der Anwendung, diese Lebensdauer zu erreichen, indem optionale Funktionalität der Anwendung – falls nötig – deaktiviert wird. So kann für eine gegebene Lebensdauer eine annähernd optimale Anwendungsqualität unter den vorhandenen Energiebeschränkungen erreicht werden.

Weiterhin erlaubt *Levels* mit Hilfe eines verteilten Algorithmus, über die gesamte Lebensdauer des Sensornetzes die durchschnittliche Anwendungsqualität im gesamten Netz nahezu konstant zu halten.

Bei der Entwicklung des Frameworks wurde besonders darauf geachtet, dass die entwickelten Konzepte auf ressourcenarmen Sensorknoten umgesetzt werden können. Deshalb wird mit einer modularen Architektur – ähnlich wie beim Sensornetz-Betriebssystem TinyOS [Hill et al. 2000] – nur die Funktionalität des Frameworks auf den Sensorknoten installiert, die für die Anwendung tatsächlich benötigt wird. Außerdem werden so viele Optimierungen und Verarbeitungsschritte wie möglich zur Compile-Zeit durchgeführt, um den Aufwand zur Laufzeit zu minimieren.

Diese Arbeit wurde im Rahmen des TinyCubus-Projekts [Marrón et al. 2005a; Marrón et al. 2005b] durchgeführt, das das Ziel hat, ein generisches rekonfigurierbares Framework für Sensornetze zu entwickeln. Bei dieser Arbeit handelt es sich um das Teilprojekt des Tiny Cross-Layer Framework. Weitere Teile von TinyCubus sind das Tiny Data Management Framework, das Softwarekomponenten zur Laufzeit adaptiert, und die Tiny Configuration Engine, die Topologieinformationen sammelt und Programmupdates im Netz verteilt.

In den folgenden Abschnitten werden nun die wichtigsten Punkte zu den einzelnen Teilen der Arbeit zusammengefasst. Zunächst wird in Abschnitt 2 eine Klassifikation von Cross-Layer-Interaktionen vorgestellt sowie ein kurzer Überblick über unsere Mechanismen zum Datenaustausch (*TinyXXL* und *Neidas*) gegeben. Dann wird in Abschnitt 3 *ViMem*, das virtuelle Speichersystem, kurz beschrieben. In Abschnitt 4 wird *Levels*, unser Ansatz zur Erstellung von energiebewussten Anwendungen, vorgestellt. Abschließend fasst Abschnitt 5 die Schlussfolgerungen und den Ausblick zusammen.

2. Schichtenübergreifender Datenaustausch

In Kapitel 4 der Arbeit wird zunächst eine Klassifikation von Cross-Layer-Interaktionen vorgestellt. Diese Klassifikation basiert auf der Analyse existierender Sensornetzanwendungen und deren Verwendung von Cross-Layer-Interaktionen. Es wird dabei zwischen unstrukturierten Interaktionen, deren negative Auswirkungen auf die Architektur besonders groß sind, und strukturierten Interaktionen unterschieden, die zumindest teilweise die Modularität erhalten.

Unstrukturiert ist das Zusammenfügen eigentlich logisch getrennter Komponenten, die Ersetzung von Systemkomponenten und die Verwendung globaler Variablen. Logisch getrennte Schichten oder Komponenten werden häufig zusammengefasst, weil sie so einfacher interagieren können. Dies verringert jedoch die Modularität und erschwert die Wartbarkeit. Systemkomponenten werden oft durch anwendungsspezifische Varianten ersetzt, wenn die ursprünglichen Versionen nicht die benötigten Schnittstellen oder Eigenschaften bereitstellen. Besser wäre es hier allerdings, zusätzliche Komponenten mit der gewünschten Funktionalität einzuführen, da sonst auch das Verhalten anderer Systemkomponenten beeinflusst werden kann. Globale Variablen sind schließlich zu meiden, weil es unklar bleibt, welche Abhängigkeiten zu den einzelnen Modulen bestehen.

Als strukturierte Cross-Layer-Interaktionen werden dagegen Funktionsaufrufe und Datenaustausch bezeichnet, wobei sich letzterer noch untergliedern lässt in Parametrisierung und die gemeinsame Nutzung von Daten. Interaktionen, die auf Funktionsaufrufen basieren, werden von Programmiersprachen wie nesC [Gay et al. 2003] bereits recht gut unterstützt. Sie sind in TinyOS zwischen allen Komponenten möglich – unabhängig davon, zu welcher Schicht sie gehören. Mangels Alternativen werden Funktionsaufrufe auch dazu eingesetzt, Daten auszutauschen. Dies stellt jedoch eigentlich eine separate Interaktionsform dar. Dabei bedeutet Parametrisierung, dass gezielt das Verhalten einer Komponente beeinflusst wird, indem andere Komponenten die von ihr definierten Parameter setzen. Die gemeinsame Nutzung von Daten dient dagegen dazu, die internen Daten für andere Schichten lesbar zu machen.

Da Datenaustausch häufig für Cross-Layer-Interaktionen eingesetzt wird und aktuell verwendete Programmierabstraktionen dafür nicht adäquat sind, weil sie die Kopplung der beteiligten Komponenten unnötig stark erhöhen, unterstützt *TinyXXL* diese Form der Interaktion explizit. Es entkoppelt interagierende Komponenten, indem die Daten nur indirekt über einen zentralen Datenspeicher (das *TinyStateRepository*) ausgetauscht werden. Das *TinyStateRepository* stellt einen Publish/Subscribe-Mechanismus zur Verfügung, über den Nutzer von Daten benachrichtigt werden, wenn sich diese ändern. Um den Laufzeitaufwand gering zu halten, erfolgt die Anmeldung dazu statisch beim Kompilieren.

Eine Komponente, die Daten anbietet oder benötigt, spezifiziert eine entsprechende Abhängigkeit. Diese Abhängigkeit wird dann von einem Prä-Compiler aufgelöst. Beim Kompilieren stellt *TinyXXL* sicher, dass alle benötigten Daten vorhanden sind, dass nur tatsächlich genutzte Daten bereitgestellt werden und dass dieselbe Art von Daten nur ein einziges Mal gesammelt wird. So wird vermieden, dass unnötig Energie verbraucht wird für das Sammeln und Bereitstellen von Daten (z. B. zum Ermitteln der Nachbarknoten). Dies wird dadurch realisiert, dass gemeinsam benutzte Daten nur geschrieben werden können, wenn der Zugriff in einem bedingten Anweisungsblock erfolgt.

Die Prüfung der Datenabhängigkeiten erlaubt es, automatisch optimierte Anwendungen aus wiederverwendbaren Komponenten zu erstellen. Eine aufwendige manuelle Optimierung der Anwendung ist damit oft nicht erforderlich. Bei den Datenabhängigkeiten können zudem neben der Art der Daten auch nicht-funktionale Eigenschaften spezifiziert werden, um z. B. die Kosten beim Datensammeln zu minimieren oder nur solche Daten zu verwenden, die eine bestimmte Genauigkeit haben.

Mit so genannten virtuellen Daten erlaubt *TinyXXL* außerdem, auf Daten zuzugreifen, die in der geforderten Form nicht im Speicher vorhanden sind; sie werden erst beim

Zugriff dynamisch aus anderen Daten erzeugt. Damit kann auf mehr Daten als nur die internen Rohdaten der Komponenten zugegriffen werden und ähnliche Daten können leicht aus vorhandenen abgeleitet oder konvertiert werden, ohne zusätzliche sammeln und speichern zu müssen.

Insbesondere mit der automatischen Optimierung und seinen virtuellen Daten geht *TinyXXL* über existierende Ansätze für Cross-Layer-Frameworks hinaus [Conti et al. 2004; Kumar et al. 2006; Su and Lim 2006]. Im Unterschied zu diesen System konzentriert sich *TinyXXL* nicht nur auf den Datenaustausch zwischen den Komponenten eines einzelnen Sensorknotens, sondern unterstützt auch den Austausch von Daten mit Nachbarknoten. So können auch Daten anderer Knoten ohne großen Entwicklungsaufwand in die Anwendung integriert werden – unabhängig davon, von welcher Schicht oder Komponente sie auf dem Ursprungsknoten bereitgestellt werden. Die zentrale Komponente dieses Ansatzes ist *Neidas*, ein Algorithmus zum Datenaustausch mit Nachbarknoten. Dieser Algorithmus ist für heterogene Netze geeignet, in denen die Knoten nicht im Voraus wissen, welche Daten ihre Nachbarn benötigen, und in denen sich die Topologie häufig ändern kann. Er nutzt die Broadcast-Eigenschaften der Funkkommunikation aus und baut auf einem Verfahren namens „Polite Gossiping“ auf, um die Zahl der übertragenen Pakete zu minimieren.

Bei Polite Gossiping [Levis et al. 2004b] senden Knoten Pakete nur, wenn sie nicht schon dieselben Information von einer gewissen Anzahl an Nachbarn erhalten haben. In *Neidas* wird dieser Mechanismus dazu eingesetzt, um sowohl die Anzahl der Datenanforderungen als auch der Datenpakete selbst zu reduzieren. Dadurch, dass Daten periodisch angefordert und gesendet werden, reagiert der Algorithmus auf Veränderungen in der Topologie. Knotenausfälle werden über einen Soft-State-Ansatz behandelt, sodass nur diejenigen Daten gesendet werden, die auch tatsächlich von Nachbarknoten benötigt werden.

Weiterhin setzt die Implementierung von *Neidas* Piggybacking ein, um bei den vergleichsweise kleinen Datenmengen den zusätzlichen Aufwand für den Paket-Kopf und die Präambel auf der MAC-Schicht einzusparen. So können die Nachbarschaftsdaten an die regulären Pakete der Anwendung angehängt werden, ohne dass dort Änderungen erforderlich wären. Nur falls die Anwendung über einen längeren Zeitraum kein geeignetes Paket schickt, wird ein separates Paket für die Nachbarschaftsdaten versendet. Insbesondere wenn Low-Power-Listening [Polastre et al. 2004] eingesetzt wird, bei dem die Präambel verlängert wird, um die Empfänger zeitweise abschalten zu können, kann durch Piggybacking der Aufwand zum Datenaustausch beträchtlich reduziert werden.

Die Evaluation in Kapitel 4 zeigt, dass komplexe Anwendungen mit *TinyXXL* erstellt werden können und dass sowohl der Entwicklungs- als auch der Laufzeitaufwand für Anwendungen mit *TinyXXL* recht gering ist. Außerdem zeigt die Evaluation, dass *Neidas* – verglichen mit einfachen Implementierungen von push- und pull-basierten Ansätzen sowie mit Hood [Whitehouse et al. 2004] – die übertragene Datenmenge erheblich reduzieren kann. Insbesondere bei dicht besetzten Netzen profitiert *Neidas* von Polite Gossiping und verringert die von jedem Knoten zu übertragenden Daten.

3. Datenspeicherung in virtuellem Speicher

Kapitel 5 beschreibt *ViMem*, ein System für flash-basierten virtuellen Speicher auf Sensorknoten. *ViMem* berücksichtigt die besonderen Gegebenheiten der Sensorknoten und ermöglicht es so, virtuellen Speicher effizient in Sensornetzen einzusetzen.

Flashspeicher ist auf typischen Sensorknoten viel größer als Arbeitsspeicher (512 KB im Vergleich zu 4 KB bei Mica2-Knoten). Allerdings sind die Zugriffskosten größer: Es können nur ganze Seiten von mehreren Hundert Bytes gelesen oder geschrieben werden. Insbesondere Schreiben erfordert einen hohen Energieaufwand und hat eine große Latenz. Anstatt – wie im RAM – eine Variable in wenigen Prozessorzyklen zu schreiben, sind beim Flashspeicher dafür mehrere Millisekunden erforderlich. Nichtsdestotrotz kann Flashspeicher als sekundärer Speicher für ein virtuelles Speichersystem verwendet werden. In aktuellen Betriebssystemen für Sensornetze [Hill et al. 2000; Han et al. 2005; Dunkels et al. 2004; Abrach et al. 2003] gibt es jedoch keine Unterstützung für virtuellen Speicher. Eine solche Funktionalität wurde erst parallel zu unserer Arbeit von t-kernel [Gu and Stankovic 2006] demonstriert, das im Unterschied zu *ViMem* allerdings nicht die Anordnung im Speicher verändert.

Wichtige Eigenschaften von Sensornetzen beeinflussen den Entwurf von *ViMem*. Zunächst wird in Sensornetz-Betriebssystemen wie TinyOS [Hill et al. 2000] der meiste Speicher statisch zugewiesen. Da nur eine Anwendung gleichzeitig ausgeführt wird, ist das Verhalten besser vorhersehbar als in Multitasking-Systemen. Zudem sind Zugriffe auf den virtuellen Speicher nur für Variablen notwendig, nicht aber bei jedem einzelnen Prozessorbefehl, da der Programmspeicher vom Arbeitsspeicher getrennt ist. Weiterhin ist Simulation zu einem unverzichtbaren Bestandteil der Programmentwicklung geworden [Titzer et al. 2005], so dass sie auch für Optimierungen ausgenutzt werden kann. Außerdem enthalten typische Sensorknoten keinerlei Hardwareunterstützung für virtuellen Speicher. Deshalb muss das ganze System in Software implementiert werden. Schließlich unterscheidet sich das Verhalten von Flashspeicher stark von anderen Speicherarten und – wie oben beschrieben – sind insbesondere Schreibzugriffe kostenintensiv. Deshalb müssen solche Zugriffe nach Möglichkeit minimiert werden.

Da die Verzögerung eines Variablenzugriffs sehr groß ist, wenn die entsprechende Seite zunächst vom Sekundärspeicher geladen werden muss, können nicht alle Variablen im virtuellen Speicher platziert werden. Insbesondere solche Variablen, die in zeitkritischen Funktionen wie Interruptbehandlungsroutinen verwendet werden, sollten weiterhin immer im Arbeitsspeicher sein. Aus diesem Grund bietet *ViMem* die Möglichkeit, die Variablen, die in den virtuellen Speicher transferiert werden sollen, explizit bei ihrer Deklaration durch ein entsprechendes Attribut zu kennzeichnen.

ViMem nutzt Simulationsläufe, um zu ermitteln, welche Variablen und Teile komplexer Variablen – so genannte Datenelemente – häufig gemeinsam benutzt werden. Auch wenn die tatsächliche Zugriffsreihenfolge in der Praxis davon abweichen kann, kann dadurch die Anzahl der Seitenfehler reduziert werden. Allerdings ist das Problem, eine Speicheranordnung mit der minimalen Anzahl Seitenfehler zu finden, NP-vollständig [Gupta 1991] – selbst wenn die genaue Zugriffsreihenfolge zur Laufzeit bekannt ist. Deshalb verwendet *ViMem* eine Heuristik, um die Anordnung im Speicher zu bestimmen.

Bei dieser Heuristik wird ein Graph aufgebaut, bei dem jeder Knoten einem Datenelement entspricht. Seine Kanten beziehen sich dagegen auf die Nähe zwischen den einzelnen Datenelementen, d. h. ihre gemeinsame Verwendung. Immer wenn auf ein Datenelement zugegriffen wird, werden das Gewicht des Knotens sowie die Kantengewichte zu den unmittelbar zuvor zugegriffenen Elementen inkrementiert. Wären diese Elemente auf einer anderen Speicherseite, würde potentiell ein Seitenfehler auftreten und eine Speicherseite müsste nachgeladen werden.

Anschließend werden die Elemente gruppiert, die häufig gemeinsam verwendet werden, d. h. deren mit den Knotengewichten normalisierte Kantengewichte besonders groß sind. Durch die Normalisierung der Kantengewichte können sowohl Gruppen entstehen, die sehr häufig verwendet werden, als auch solche, die nur selten benutzt werden. Sie werden dann auf Speicherseiten platziert, wobei zwei Klassen von Seiten benutzt werden. Auf der einen Klasse werden hauptsächlich die Elemente platziert, die häufig geschrieben werden, während auf der anderen solche angeordnet werden, die vorwiegend gelesen werden. Die letzteren Seiten müssen somit nur selten auf den Flashspeicher zurückgeschrieben werden, auch wenn sie aus dem Arbeitsspeicher entfernt werden müssen.

Die Implementierung von *ViMem* für Mica2-Knoten nutzt die im Flash-Chip enthaltenen SRAM-Puffer, um die Anzahl der tatsächlichen Schreibzugriffe auf den Flashspeicher zu reduzieren. Dies ist deshalb möglich, da die Konsistenz und Persistenz des virtuellen Speichers nach einem unerwarteten Neustart nicht gewährleistet sein muss; die Daten der Puffer können also verloren gehen.

In der Evaluation dieses Kapitels wird mit Simulationen gezeigt, dass *ViMem* die Anzahl der Zugriffe auf den Flash-Speicher stark verringern kann im Vergleich zu einem Ansatz, bei dem die Variablen entsprechend ihrer Deklarationsreihenfolge angeordnet werden und der damit schon die natürliche Lokalität der Daten ausnutzt. Dazu müssen allerdings genügend Informationen über die Zugriffsmuster zur Verfügung stehen, um das Speicherlayout anpassen zu können. Für diese Simulationen wurden TinyDB [Madden et al. 2005] und Maté [Levis et al. 2005] als Beispiele für komplexe Anwendungen so abgeändert, dass sich Teile ihrer Variablen im virtuellen Speicher befinden. Weiterhin wird untersucht, wie sich die Anzahl der Speicherseiten im Arbeitsspeicher sowie die Gesamtgröße der Daten im virtuellen Speicher auf die Anzahl der Seitenfehler und den Energiebedarf auswirken.

4. Abstraktionen und Algorithmen für energiebewusste Anwendungen

Kapitel 6 beschreibt *Levels*, den Teil des Frameworks, der Energieaspekte in die Systemsoftware verschiebt. *Levels* sorgt dafür, dass eine vorgegebene Lebensdauer erreicht wird, wobei die Anwendungsqualität maximiert werden soll. Es stellt dazu die Programmierabstraktion der Energiestufen bereit. Diese Stufen enthalten optionale Programmteile, die nicht unbedingt erforderlich sind, um eine Basisfunktionalität der Anwendung zu gewährleisten. Beispielsweise kann in einem dünn besetzten Netz ein Knoten seine energieintensiven Sensoren abschalten, wenn er dadurch die Netzwerkkonnektivität erhalten kann. Somit ist er – trotz reduzierter Funktionalität – nützlicher, als wenn er komplett ausfallen würde, auch wenn er selbst keine Sensordaten mehr sammeln kann.

Mögliche Anwendungen findet man beispielsweise bei der Strukturüberwachung von Brücken [Kim et al. 2007], wo Batterien nur bei regelmäßigen Inspektionen alle paar Jahre ersetzt werden können [Marrón et al. 2005c]. Dadurch wird die geforderte Lebensdauer der Knoten bestimmt. Weiterhin sollte hier kein Knoten vorzeitig ausfallen, weil die Topologie typischerweise sehr dünn besetzt ist. Durch Abschalten der Sensoren kann bei dieser Anwendung ein beträchtlicher Teil der benötigten Energie eingespart werden.

Bei einer anderen Anwendung, dem Überwachen des Mikroklimas von Bäumen [Tolle et al. 2005], ist die Netzwerkdichte so groß, dass die Konnektivität auch bei Knotenausfällen erhalten bleiben könnte. Dennoch kann auch hier davon profitiert werden, wenn jeder Knoten die gesamte Lebensdauer erreicht. In dieser Anwendung könnte beispielsweise das recht energieaufwendige Speichern einer zusätzlichen Kopie der Daten im Flashspeicher unterbleiben, wenn diese sowieso zur Basisstation gesendet werden. Dadurch gehen vielleicht einzelne Werte verloren, wenn ein Übermittlungsfehler auftritt, aber die räumliche Auflösung der Sensorwerte bleibt dennoch für die geforderte Lebensdauer erhalten, da dies nur einzelne Pakete betreffen dürfte.

Eine Energiestufe enthält alle Befehle, die zusammen deaktiviert werden können. Beispielsweise ist dies bei der Strukturüberwachung alles, was mit dem Erfassen von Sensorwerten zusammenhängt. Um Programmcode einer Energiestufe hinzuzufügen, muss für diesen einfach mit einer bedingten Anweisung geprüft werden, ob die Energiestufe aktiv ist. Alle Energiestufen einer Anwendung bilden zusammen einen Stapel, bei dem sie – beginnend mit der obersten Stufe – deaktiviert werden können. Wenn eine Stufe aktiv ist, sind auch alle darunterliegenden aktiv. Die unterste Stufe wird von dem gesamten Programmcode gebildet, der nicht explizit einer höheren Stufe hinzugefügt wurde. Diese Energiestufe ist immer aktiv.

Dieses Konzept passt sich gut in die modulare Entwicklung von Anwendungen mit komponentenbasierten Programmiersprachen wie nesC ein. So können in einzelnen Komponenten Energiestufen definiert werden, die dann beim Erstellen der Anwendung kombiniert werden. Für jede dieser Energiestufen kann der Anwendungsentwickler einen Nützlichkeitswert angeben. Dieser legt fest, wie groß der Gewinn an Funktionalität im Vergleich zu der darunterliegenden Stufe ist. So steigt die Nützlichkeit vielleicht stärker an, wenn in einer Stufe Sensordaten gesammelt werden sollen, während der zusätzliche Gewinn einer lokalen Speicherung dieser Daten kleiner ausfällt.

Levels benötigt zur Laufzeit Informationen über den Energiebedarf der verschiedenen Energiestufen. Um diesen abschätzen zu können, wird mit Hilfe eines Simulators ein Energieprofil aller optionalen Code-Blöcke der Energiestufen erstellt. Dazu können beispielsweise Testfunktionen aus dem Modultest wiederverwendet werden, da auch hierfür jede Funktion unabhängig von der gesamten Anwendung ausgeführt werden sollte. Diese Energiemessung ermöglicht es, auch den Energiebedarf von asynchron ausgeführtem Code (z. B. TinyOS-Tasks, Interruptbehandlungsroutinen) korrekt der richtigen Energiestufe zuzuordnen. Außerdem erlaubt sie – indem mehrere Messungen durchgeführt werden – eine Berechnung sowohl der Energie, die von der entsprechenden Funktion einmalig beim Aufruf benötigt wird, als auch einer Änderung im kontinuierlichen Energiebedarf (z. B. beim Aktivieren einer Hardwarekomponente).

Da typische Sensorknoten wie die Mica2 keine direkte Beobachtungsmöglichkeit des Energieverbrauchs anbieten, muss dieser über die Batteriespannung berechnet werden. Diese Spannung fällt mit der Kapazität ab. Obwohl es sich dabei nicht um den Hauptfokus von *Levels* handelt, ist eine Abbildung der Spannung auf die verbleibende Batteriekapazität notwendig, um ein funktionierendes System zu bilden. Für *Levels* wurde deshalb ein recht einfaches, aber effizientes Modell mit Hilfe von Messungen erstellt, das jeden Spannungswert auf einen Energiewert abbildet. Außerdem gibt dieses Modell die erwartete Genauigkeit für jeden Wert an, da sich diese für verschiedene Spannungswerte stark unterscheiden kann. Um die Berechnungen zur Laufzeit zu vereinfachen und dennoch korrekte Ergebnisse zu erhalten, wird sowohl bei der Erstellung des Modells als auch zur Laufzeit von einem Energieverbrauch für ein Codestück ausgegangen, der unabhängig von der aktuellen Versorgungsspannung ist.

Zur Laufzeit wird mit Hilfe des Batteriemodells die verbleibende Kapazität und der aktuelle Gesamtverbrauch des Knotens berechnet. Außerdem wird mit den Informationen des Energieprofils der Bedarf der einzelnen Stufen ermittelt. Aufbauend auf diesen Daten kann dann eine optimale Zuweisung von Energiestufen berechnet werden.

Dazu wird ein Problem der linearen Programmierung mit Hilfe des Simplex-Algorithmus gelöst. Es soll dabei die Zeitdauer in den Energiestufen berechnet werden, die ihre durchschnittliche Nützlichkeit über die Zeit maximiert. Dies soll unter Beachtung der folgenden Randbedingungen geschehen: Die Zeitdauer in allen Energiestufen soll der geforderten verbleibenden Lebensdauer des Knotens entsprechen, und es soll höchstens so viel Energie benötigt werden, wie auch zur Verfügung steht.

Da *Levels* die Optimierungen lokal auf den einzelnen Knoten durchführt, schwankt die netzwerkweite Gesamtqualität der Anwendung stark, weil viele Knoten zu ähnlichen Optimierungsergebnissen kommen und deshalb dieselben Energiestufen zur selben Zeit wählen. Um diese Qualität annähernd konstant zu halten, bietet *Levels* verschiedene Mechanismen an. Erstens kann der Anwendung die Entscheidung überlassen werden, welche Stufe aus dem lokalen Optimierungsergebnis ausgewählt werden soll. Zu diesem Zweck kann sie ihre eigene, anwendungsspezifische Koordination durchführen. Zweitens kann eine Stufe zufällig aus dem lokalen Ergebnis gewählt werden. Dies sorgt für eine gleichmäßigere Verteilung über die Zeit. Schließlich bietet *Levels* eine verteilte Heuristik an, die die Stufenzuweisungen lokal in der Nachbarschaft koordiniert. Sie kann außerdem dazu benutzt werden, in dichten Netzen Knoten zeitweise zu deaktivieren, wobei auch hierbei die Anwendungsqualität konstant gehalten werden soll.

Bei diesem verteilten Ansatz versucht jeder Knoten, die Energiestufen aus seinem lokalen Ergebnis so über die Zeit verteilt zuzuweisen, dass die Abweichung vom durchschnittlichen Wert in der Nachbarschaft minimiert wird. Dabei kann ein Greedy-Ansatz die lokal optimalen Zuweisungen mit geringem Aufwand bestimmen. Das Ergebnis sendet jeder Knoten dann an seine Nachbarn, die dieses Ergebnis bei der Durchschnittsberechnung für ihre eigene Zuweisung berücksichtigen. Es handelt sich um eine Heuristik, die nicht notwendigerweise das globale Optimum findet, da jeder Knoten nur seine eigenen Zeitintervalle beeinflussen kann und die Werte seiner Nachbarn als gegebene Randbedingungen hinnehmen muss.

Das Kapitel enthält eine Evaluation von *Levels*, in der sowohl mit Simulation als auch mit Hilfe von Experimenten mit realen Sensorknoten gezeigt wird, dass *Levels* die angeforderte Lebensdauer recht genau erreicht und somit die Qualität der Anwendung fast optimal ist. Wie die Experimente zeigen, werden dazu allerdings recht genaue Informationen über die Batterieentladung benötigt. Weiterhin wird eine komplexe Anwendung zur Vulkanüberwachung [Werner-Allen et al. 2006a] so abgeändert, dass sie *Levels* nutzt. Für die Integration von *Levels* waren nur geringe Änderungen am Programmcode der Anwendung notwendig. Obwohl diese Anwendung von externen Ereignissen abhängt, erreicht auch sie die angeforderte Lebensdauer. Außerdem wird mit Simulationen dargestellt, dass sich durch die verteilte Koordination der Zuweisungen die Abweichung von der durchschnittlichen Anwendungsqualität erheblich verringern lässt. Schließlich zeigt die Evaluation, dass der Laufzeitaufwand von *Levels* vernachlässigbar klein ist.

5. Zusammenfassung und Ausblick

Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick auf Forschungsfragen, die sich aus dieser Arbeit ergeben (siehe Kapitel 7).

Zusammenfassend lässt sich sagen, dass das entwickelte Cross-Layer-Framework einen breiten Bereich von Cross-Layer-Interaktionen behandelt, auch wenn dadurch nicht alle negativen Auswirkungen dieser Interaktionen beseitigt werden können. In manchen Aspekten ist das Framework sogar allgemeiner als das ursprüngliche Ziel. Beispielsweise können mit dem Algorithmus zur Anforderung von Daten der Nachbarknoten nicht nur klassische Cross-Layer-Daten ausgetauscht werden, sondern auch z. B. Anwendungsdaten. Ähnlich hilft *Levels* nicht nur einige Cross-Layer-Interaktionen zu vermeiden, sondern bietet davon unabhängige Mechanismen an, um eine gegebene Lebensdauer zu erreichen. Daher verringert das Framework nicht einfach die Nebenwirkungen von Cross-Layer-Interaktionen, sondern hilft dem Anwendungsentwickler, effiziente Sensornetzanwendungen zu erstellen.

Zukünftige Forschungsfragen betreffen sowohl eine Ergänzung des Frameworks um andere Arten von Cross-Layer-Interaktionen und mögliche Erweiterungen der einzelnen Teile als auch die erneute Betrachtung von Entwurfsentscheidungen abhängig von der weiteren Entwicklung der Sensornetze-Forschung. Aller Voraussicht nach werden Cross-Layer-Interaktionen bei billigen, ressourcenbeschränkten Geräten, die drahtlos miteinander kommunizieren, auch in Zukunft wichtig bleiben.

1. Introduction

1.1. Motivation

In the last few years wireless sensor networks have been introduced to unobtrusively collect data from the physical world. The nodes of such a network – so-called sensor nodes – are able to sense physical data, process this data within the network, and transmit it using wireless multi-hop communication. It is widely believed that sensor networks will have a huge potential in the future [Estrin et al. 1999; Römer and Mattern 2004; Marrón et al. 2006b].

Several promising application scenarios for sensor networks have been described in the literature. Much of the research in the early years has focused on military applications [Arora et al. 2004] and on environmental monitoring. For example, sensor networks have been used to get information about the micro-climate of redwood trees [Tolle et al. 2005], to monitor animals in their natural habitat [Szewczyk et al. 2004; Juang et al. 2002], and to detect volcanic eruptions [Werner-Allen et al. 2006a]. For these environmental monitoring applications sensor nodes help to unobtrusively observe physical phenomena at a previously unknown fidelity and scale. Although there are some exceptions, most of the applications in this domain just gather, aggregate, and forward data without doing more complex processing in the network.

Besides these scientific applications, other – more commercial – applications have already emerged. For instance, sensor networks have been proposed for industrial monitoring where they have been used to predict equipment failure from vibration patterns [Krishnamurthy et al. 2005]. Compared to previous solutions that used wired sensors or had employees with mobile devices read out the data, sensor networks reduce costs greatly. Likewise, another promising application is structural health monitoring of buildings and bridges [Xu et al. 2004; Marrón et al. 2005c; Kim et al. 2007]. Although maintenance personnel still has to manually inspect the machines and structures from time to time, the use of sensor networks allows to detect defects early at reduced costs. Therefore, they can be repaired before large damage occurs.

Other applications include precision agriculture to prevent deceases from spreading and to increase the harvest as well as animal production [Langendoen et al. 2006; Wark et al. 2007]. For example, a sensor network can be used to provide data on humidity and temperature close to the crop or measure soil moisture. This information allows to infer when to apply fertilizer and when to move cattle to another pasture, for example.

Finally, in intelligent transport scenarios cars can be regarded as sensor nodes [Reichardt et al. 2002; Marrón et al. 2005b]. In these applications the cars send, for example, messages to warn other vehicles of dangerous situations or to assist in lane merging.

In most of these examples sensor nodes are powered from batteries. In order to create inexpensive devices that operate for a long time without human intervention sensor nodes are strictly resource-constrained [Hill et al. 2004]. Because of the limited battery capacity especially the amount of energy available is a major constraint. The energy restrictions have lead to the use of small, efficient microcontrollers. Therefore, typically not just energy is a limiting factor but also the amount of main memory available and – to a lesser extent – the computational capabilities.

The resource restrictions of sensor nodes lead to a need for optimizations in sensor network applications. The goal of such optimizations is to conserve energy (e.g., by reducing radio communication) or to make the memory footprint in RAM smaller (e.g., by avoiding redundant data). Only with such optimizations it is possible to run a sensor node for weeks or even months with a single set of batteries or to develop complex software systems for hardware platforms with only a few kilobytes of RAM. One technique for such optimizations is to use cross-layer interactions, where – in contrast to a strictly layered architecture [Garlan and Shaw 1993] – software layers interact more closely. Cross-layer interactions are widely used in sensor networks [Levis et al. 2004a] and are often regarded as a necessity [Goldsmith and Wicker 2002]. In addition, cross-layer interactions offer the possibility of dealing with the special properties of wireless networks that cannot be handled well by strictly layered architectures. For example, this can be coping with variations in link quality or adjusting the transmission strength, which influences the radio range, the number of collisions, and energy consumption.

As described by Kawadia and Kumar [Kawadia and Kumar 2005], cross-layer interactions can have a negative effect on desirable properties of the software architecture such as modularity. For example, if cross-layer interactions are not performed in a controlled fashion, it might not be possible to exchange a module without major changes to others. This reduces modularity and the reusability of software components. In the extreme, the system becomes a monolithic piece of code that is hard to develop and impossible to maintain. Thus, if cross-layer interactions are needed, they have to be used deliberately. Therefore, in this thesis we propose a set of programming abstractions and the corresponding system software that help in dealing with these interactions.

The resulting framework assists the developer in three ways when applying cross-layer interactions. First, it aims to alleviate the negative effects on the software architecture by decoupling interacting software components. For example, this allows to deal with the special properties of wireless communication. Second, the framework tries to reduce the need for cross-layer interactions by relaxing the constraints that lead to cross-layer interactions. Finally, it deals explicitly with these constraints inside the system software so that the application can achieve its target lifetime without additional cross-layer interactions.

1.2. Contribution

This thesis has several contributions. First, based on an analysis of existing sensor network applications it presents a classification of the cross-layer interactions found there. Second, it describes a framework with its algorithms and heuristics that pursues three strategies to alleviate the negative side-effects of cross-layer interactions. As described above, the first strategy tries to preserve modularity and reusability by decoupling the interacting components. In this part of the framework we focus on some particular forms of cross-layer interactions that deal with exchanging data among layers. The second strategy aims to avoid some cross-layer interactions by providing an abstraction that addresses one of the main causes why they are used in the first place. In particular, this abstraction tries to alleviate the memory limitations by providing an optimized virtual memory system. Finally, the third strategy deals explicitly in the framework with another one of these causes: energy limitations. Using the abstraction of a so-called energy level it allows to meet user-defined lifetime goals while providing good application quality. This way it does not only reduce the effort of the application developer but also makes some cross-layer interactions unnecessary. However, even with this part of the framework complex applications cannot do without all cross-layer interactions if they have to operate efficiently.

With these three different strategies our cross-layer framework follows a much broader approach than other frameworks. Typically, existing cross-layer frameworks for sensor networks and mobile ad-hoc networks (MANETs) provide means to facilitate cross-layer interactions [Conti et al. 2004; Kumar et al. 2006; Su and Lim 2006] but do not try to address their causes. The following paragraphs give some details about the parts of our cross-layer framework that employ its three strategies.

To decouple software components the cross-layer framework provides mechanisms to exchange data among several layers. Such functionality helps to reduce memory consumption since redundant data does not have to be kept in RAM by several layers. In addition, energy consumption is possibly also reduced because data does not have to be gathered several times by, for example, sending messages if it is already available on another layer. Finally, this mechanism allows the software developer to create a cross-layer network stack that can better deal with the properties of wireless communication. Unlike most existing systems supporting data exchange [Conti et al. 2004; Kumar et al. 2006; Su and Lim 2006; Köpke et al. 2004] our approach automatically performs many optimizations at compile-time and avoids, for example, having meta-data in RAM.

In order to create a comprehensive system for both intra-node and inter-node data sharing, the cross-layer framework includes an efficient protocol to share data with neighboring nodes. Data from neighbors allows for potentially new optimizations in the applications without requiring much additional development effort. Other approaches that provide programming abstractions for neighborhood communication [Whitehouse et al. 2004; Welsh and Mainland 2004; Mottola and Picco 2006] either include just simple algorithms to actually share the data or leave the design of such an algorithm to the application developer.

To address memory limitations differently and to make many cross-layer interactions unnecessary, the cross-layer framework includes a virtual memory system that increases the amount of memory available by placing currently unused data in flash memory. Therefore, it relaxes the strict memory constraints that are the cause of many cross-layer interactions. It addresses the specific properties of the sensor nodes and – unlike existing virtual memory solutions for sensor networks [Gu and Stankovic 2006] – tries to optimize the memory layout. Thus, it reduces the overhead of such a virtual memory solution, concerning both access latency and energy consumption. Other approaches that optimize memory layouts either focus on restructuring code [Muchnick 1997; Hatfield and Gerald 1971; Hartley 1988], which is stored separately on typical sensor nodes, or do not take into account the characteristics of flash memory [Gupta 1991; Stamos 1984]. Furthermore, the virtual memory system allows to keep more cross-layer data in memory and, therefore, avoids energy-expensive data gathering.

Finally, to explicitly address energy constraints the cross-layer framework includes mechanisms that measure the energy consumption of some code and adjust the functionality of the sensor nodes. It allows to meet a user-defined lifetime goal while providing the best quality possible. Most existing approaches are not targeted to the resource limited platforms of sensor nodes [Flinn and Satyanarayanan 1999; Zeng et al. 2002], provide only limited mechanisms to adjust application quality [Madden et al. 2005], or focus on *maximizing* lifetime instead of meeting a target lifetime at a good quality [Cardei and Wu 2006; Cerpa and Estrin 2002; Giusti et al. 2007]. Meeting a lifetime goal for the sensor network can be useful in many classes of applications such as, for example, structural health monitoring and environmental monitoring.

Like the virtual memory system, this part of the framework tries to avoid some cross-layer interactions. However, instead of relaxing the constraints that cause them, it tries to reduce the burden on the developer when creating energy-efficient applications. This way it moves energy concerns into the system software and the developer is less likely to apply cross-layer interactions for energy efficiency. In addition, using minimal data exchange among nodes, a distributed algorithm can assign node schedules that keep the overall quality of the application almost constant.

The major contributions of this thesis have been published in international scientific conferences [Lachenmann et al. 2005; Lachenmann et al. 2006; Lachenmann et al. 2007a; Lachenmann et al. 2007b; Lachenmann et al. 2007c].

1.3. Structure

The rest of this thesis is structured as follows. Chapter 2 introduces the context in which this work has been done. It includes an overview of wireless sensor networks and their applications, an introduction to cross-layer interactions, and a description of the TinyCubus project, of which the cross-layer framework is part.

Chapter 3 summarizes the functionality of the framework, gives an overview of its individual parts, and presents properties of the overall design.

Following this overview, the next chapters describe the three parts of the cross-layer framework in more detail:

First, Chapter 4 presents our analysis of cross-layer interactions in existing applications. In addition, it describes the abstractions and algorithms for data exchange between both the software components of a single node and neighboring nodes. The chapter includes an evaluation that shows the efficiency of this approach.

Second, Chapter 5 presents and evaluates the cross-layer framework's virtual memory system. This system uses a heuristic to optimize the layout of variables in virtual memory. The evaluation results show that for typical sensor network application the heuristic can greatly reduce the access costs to virtual memory.

Third, Chapter 6 describes the abstraction of a so-called energy level, which allows to adjust the functionality of the application in order to meet a user-defined lifetime goal. In addition, it includes a distributed mechanism to balance node schedules over time. Using both simulation and experiments with Mica2 nodes, the chapter shows that this approach is able to accurately meet a lifetime goal despite many inaccuracies present on sensor nodes.

Finally, Chapter 7 summarizes the contributions of this thesis and identifies possible extensions for future work.

2. Background

This chapter presents background information on the work of this thesis. It describes wireless sensor networks in more detail – including their applications and hardware properties – and states the underlying assumptions of this thesis. It then defines the term "cross-layer interaction" and gives an overview of such interactions. Finally, this chapter describes the TinyCubus project, in which our cross-layer framework has been developed.

2.1. Wireless Sensor Networks

Within the last few years there has been significant research in the area of wireless sensor networks. This section summarizes properties of typical platforms regarding both hardware and software. In addition, it states the assumptions of our system model.

2.1.1. System Properties

Hardware Platforms

As already mentioned in the introduction, sensor nodes form a wireless network using ad-hoc communication. Typically, each sensor node is equipped with a microcontroller, a radio chip, non-volatile flash memory, and application-specific sensors. In this thesis we focus specifically on the small, resource-limited, inexpensive "mote-class" devices [Hill et al. 2004]. These kinds of devices are commonly used in sensor network research; they are already commercially available from several manufacturers. Examples include the Telos nodes [Polastre et al. 2005b] and the Mica2 nodes [Hill et al. 2004]. Since the implementation of our cross-layer framework is targeted towards Mica2 nodes the following description focuses on this hardware platform.

A Mica2 node is equipped with an 8-bit RISC processor from Atmel that runs at 7.37 MHz. Its main memory consists of 4 KB of RAM. The program is stored separately in flash-based program memory, which is much larger in size (128 KB). Besides these two memories that are internal to the CPU, the Mica2 nodes – like most available sensor nodes – are equipped with an external flash memory chip of 512 KB. Flash memory can be used, for example, to store sensor readings.

The Mica2 nodes communicate via wireless transmission in an ISM band. The communication chip allows to implement even the MAC layer in software. Therefore, cross-layer interactions can be applied easily in the complete network stack.

The user can add application-specific sensor boards to the Mica2 nodes. Without such a board, the sensor node can only measure its internal battery voltage. Typical sensor boards, which are commercially available, include light, temperature, and acoustic sensors. Other boards are equipped with accelerometers and magnetometers or with humidity, barometric pressure, and seismic sensors.

Finally, the Mica2 nodes have three LEDs, which can be used for debugging purposes, and an interface to connect them to the serial port of a PC. If a node is connected to a PC, it acts as a gateway to the sensor network.

Like most sensor nodes available today, the Mica2 nodes are powered by two AA batteries. Therefore, if the node should be operational for more than a few days, energy is a precious resource. The energy consumption of the sensor node can vary significantly depending on the current states of the CPU, radio chip, sensors, and flash memory chip, which are the main energy consumers. For example, power consumption of the CPU can increase from $348\,\mu$W in its "power down" state to $22.8\,$mW in the "active" state [Landsiedel et al. 2005]. The power consumption of the radio chip is often in the same order of magnitude whereas custom sensor interface boards sometimes consume as much energy as the remaining components of the sensor node combined [Werner-Allen et al. 2006a; Kim et al. 2007].

The Mica2 nodes have been a popular research platform for several years now. They are supported by many sensor network operating systems and tools. Even though some new hardware platforms have emerged recently, their properties (e.g., the CPU speed, RAM size, battery-power, flash memory as secondary memory) are very similar. Therefore, it can be expected that the challenges addressed in this dissertation will still be present in the future. Instead of using advances in hardware for faster processors with more RAM it is more likely that these advances will lead to a reduction in cost, size, and energy consumption [Gay et al. 2003; Dunkels et al. 2004; Gehrke and Madden 2004].

Operating Systems

Several operating systems have been proposed to specifically address the challenges of sensor networks. The best-known examples are TinyOS, Contiki, SOS, and Mantis.

TinyOS [Hill et al. 2000] is probably the most popular operating system for sensor networks. It has an active community of developers and has been ported to several hardware platforms including the Mica2 nodes. Several large applications have been developed using TinyOS and have been tested in real-world deployments. TinyOS has been implemented in nesC [Gay et al. 2003], a programming language specifically

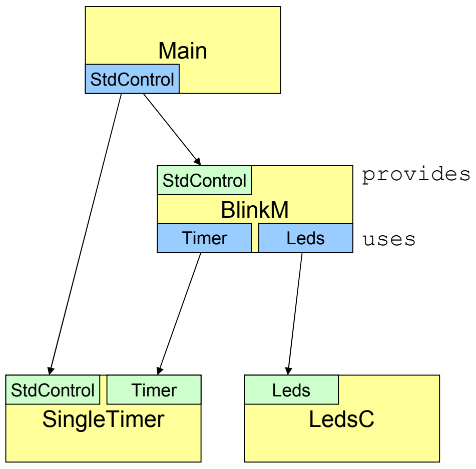


Figure 2.1.: Wiring of Blink, a simple TinyOS application

designed to support the TinyOS abstractions of components, events, and tasks.

The concurrency model of TinyOS is based on an event-driven system. All events are executed atomically with respect to other events. Therefore, to maintain the reactivity of the system longer computations are done in so-called tasks. Tasks can be suspended by events, but not by other tasks. A simple scheduler runs the tasks in FIFO order.

A TinyOS application consists of system and application components that are connected ("wired") using their interfaces. Only those components that are actually needed by the application are included in the code image. This reduces the size of both the code image and the variables in RAM. Each component can be wired to any other one. Therefore, there is no explicit notion of layers in TinyOS and cross-layer interactions can be applied more easily.

For example, Fig. 2.1 shows how a simple application is composed of several existing components. Each component like "BlinkM" specifies the interfaces that it provides ("StdControl") and those that it uses ("Timer" and "Leds"). The application is then created by wiring these dependencies to other components.

In nesC code the wirings are defined in so-called configurations. For instance, Fig. 2.2 shows the code needed to create the wiring of the sample application of Fig. 2.1. This application, "Blink" is part of the TinyOS distribution. It periodically toggles one of the node's LEDs. Each of the components that are used in this code fragment can be a configuration, which composes other components, or a module, the second type of nesC components.

A module includes the actual code that implements the interfaces provided. The code of BlinkM, the module that implements the Blink application's main functionality, is shown in Fig. 2.3. The commands of the "StdControl" interface are used for initialization and to control a timer. Whenever this timer fires (event "Timer.fired"), the node toggles its red LED using the "Leds" interface.

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;

  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;

  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

Figure 2.2.: nesC configuration for the Blink application

Although TinyOS is probably the most-widely used operating system for sensor networks, there are several alternatives.

First, Contiki [Dunkels et al. 2004] provides more flexibility than TinyOS. Instead of replacing the complete code image when installing updates – as it is required with TinyOS – Contiki allows to replace loadable modules and the kernel itself individually.

Contiki is based on an event-based kernel but includes an optional library for multi-threading. This library simplifies application development but increases memory consumption since each thread needs its own stack. Therefore, as a third approach Contiki supports so-called proto-threads [Dunkels et al. 2006] which provide a thread-like programming interface that the compiler transforms to an event-based implementation.

Second, SOS [Han et al. 2005], which is similar to Contiki, allows for dynamic replacement of modules. However, it does not include a multi-threading abstraction.

Finally, Mantis [Abrach et al. 2003] includes such an abstraction. This system has the goal of providing an easy-to-use environment for rapid prototyping. Therefore, it tries to transfer well-known abstractions like multi-threading to the field of sensor networks. In addition, it includes a layered network architecture, where several layers can be combined in a single thread.

For the implementation of the concepts presented in this thesis we have selected TinyOS as the underlying operating system. The broad range of existing applications that have been developed with this system allows to test the framework in different scenarios. In addition, with its component-based architecture it allows to apply cross-layer interactions more easily than Mantis, for example.

2.1.2. Assumptions

This thesis makes some assumptions on the classes of devices used and on properties of the network. Some of the assumptions are valid only for parts of the framework.

```
module BlinkM {
  provides interface StdControl;
  uses interface Timer;
  uses interface Leds;
}
implementation {
  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }

  command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 1000);
  }

  command result_t StdControl.stop() {
    return call Timer.stop();
  }

  event result_t Timer.fired() {
    call Leds.redToggle();
    return SUCCESS;
  }
}
```

Figure 2.3.: nesC module for the Blink application

Therefore, after describing general assumptions relevant to more than one part this section gives details on specific assumptions for the parts dealing with cross-layer data exchange, virtual memory, and energy awareness.

First of all, the framework assumes as a hardware platform small inexpensive, battery-powered sensor nodes like well-known mote-class devices [Hill et al. 2004]. Therefore, the resources on the sensor nodes, especially main memory and energy, are strictly limited. These limitations make some optimizations necessary when creating complex applications that can run for a long time from a single set of batteries. Program memory, which is assumed to be separate from main memory, is – in contrast – expected to be less constrained than RAM.

To support a wide range of applications, networks with both relatively few nodes (tens of nodes) and large numbers (hundreds or thousands) should be supported. This assumption is relevant for neighborhood data sharing and our abstraction for energy-aware applications but not for the other parts since they do not make use of network communication.

Since the components dealing with virtual memory and energy awareness rely on information from simulation, they have the assumption that the relevant properties of sensor networks are captured in the simulator. This refers especially to the costs of flash memory accesses, energy consumption of the different hardware devices, and the battery capacity available. We verified this assumption for Mica2 nodes and the Avrora simulator [Titzer et al. 2005] by measuring these properties with real hardware nodes.

To dynamically adapt the system to new properties TinyCubus assumes that software components can be replaced at runtime. In TinyOS this requires the use of a special linker running on the sensor nodes [Marrón et al. 2006a]. Moreover, the underlying hardware platform has to support writing to program memory without requiring an external programming device.

Data Exchange

The framework's part dealing with data exchange assumes that for data sharing it is sufficient if there is a single component providing a particular kind of data. In addition, it supposes that the relevant non-functional properties can be specified in a way that makes it possible to fulfill the requirements of all data users.

Our neighborhood data sharing algorithm assumes that a cheap broadcast channel is available, where nodes can overhear the messages of their neighbors. This depends on the duty cycles of the nodes and the MAC layer protocol. Commonly used protocols like B-MAC [Polastre et al. 2004] and IEEE 802.15.4 support such a broadcast channel but, for example, some TDMA-based protocols might only provide point-to-point unicast communication.

Furthermore, the neighborhood data sharing algorithm assumes that radio communication is lossy. This means that not all messages are delivered successfully on the first try. Moreover, the topology is assumed to be frequently changing and nodes may fail. Since the algorithm has been developed with these properties in mind, it would offer suboptimal performance in an unrealistic, lossless network with a static topology. In addition, the links between nodes do not have to be symmetric, i.e., even if node A can hear node B, B cannot necessarily receive packets from node A.

Finally, the neighborhood data sharing algorithm assumes that each requested data item is needed by more than one node in the neighborhood.

Data Storage in Virtual Memory

Our virtual memory system assumes that a form a of larger secondary storage is available. Its currently used memory layout algorithm has been specifically optimized to flash memory, where, for instance, writing is much more expensive than reading.

Taking into account the access delay of virtual memory, our system assumes that the processing power available is not a bottleneck on sensor nodes. As an analysis shows [Levis et al. 2004a], this is true for existing applications where the CPU is idle most of the time.

This algorithm also assumes that all data – except for the stack – is statically allocated at compile-time. Even though dynamic memory allocation is possible on some platforms, TinyOS, for example, discourages its use in order to avoid memory leaks. Therefore, in order to deal with the worst case, applications usually allocate more memory than they need most of the time. This makes it possible to optimize the memory layout because only a subset of the data is used during normal execution.

In addition, the virtual memory system assumes that variables are not accessed uniformly but that some of them are accessed more frequently than others and that there are groups of variables which are often used together. In our experience this assumption holds for almost any real-world application.

Abstractions and Algorithms for Energy-Aware Applications

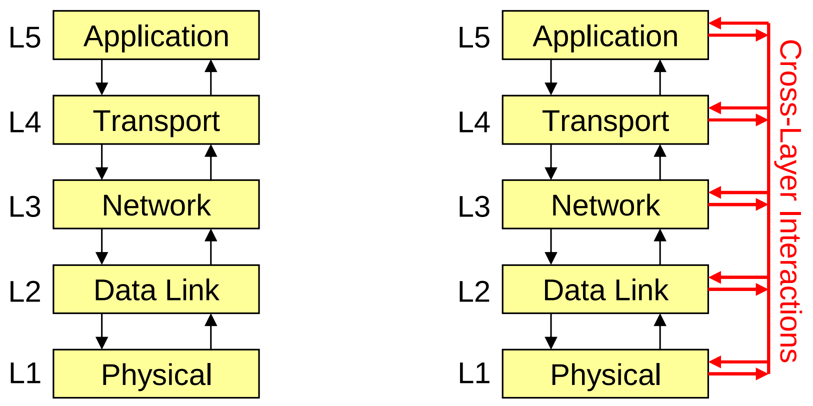
The part of the framework dealing with energy issues assumes that information about the energy used or the energy left in the batteries is available. This can be exact energy readings obtained with a battery monitoring chip or – as in the case of standard Mica2 nodes – this information can be inferred from the current battery voltage.

The applications created with this part of the framework are assumed to have some energy-expensive functionality that is not really needed for a sensor node to be useful. For example, such basic functionality can preserve network connectivity without gathering sensor data. In addition, we assume that an application consumes more energy in higher energy levels than in lower ones. This condition has to be ensured by the application developer.

Furthermore, if our distributed coordination approach of level assignments is used, we assume that there are enough nodes in the network neighborhood to balance their assignments. In addition, when using the coordination approach nodes may not be mobile since the schedule is balanced for longer time intervals.

2.2. Cross-Layer Interactions

Layered architectures have been successfully used for many years now. Most notably, layers have been employed in communication architectures but also in operating system and database design [Garlan and Shaw 1993]. For network communication the ISO OSI (Open Systems Interconnection) reference model proposes a layered architecture. Similarly, the widely used Internet network protocols form layers in the network protocol stack (see Fig. 2.4(a)) [Tanenbaum 2003].



(a) Protocol stack described by Tanenbaum [Tanenbaum 2003]

(b) Cross-layer interactions

Figure 2.4.: Layers in the network protocol stack

In such a strictly layered architecture each layer can only be accessed from the adjacent layers, i.e., the layers immediately above or below it. In addition, this interaction is limited to narrow, well-defined interfaces. For example, in a network protocol stack the layers just pass packets to the next layer without skipping layers and without providing additional means for closer interaction.

Layered architectures have several advantages [Garlan and Shaw 1993]. First, they allow to increase the level of abstraction with each layer. Therefore, for example, lower layers deal with communication between directly connected nodes whereas higher layers route packets to other networks without having to deal with issues like medium access control. Second, layered architectures support maintainability since changes are limited to a specific layer or – if its interfaces change – to at most the two adjacent ones. Finally, layered architectures support the reuse of software since layers are only loosely coupled. Several implementations can be used interchangeably if they just provide the same interfaces. This is especially interesting if these interfaces are standardized like in the OSI model. For instance, the same higher layer protocols can be used to communicate over different link layer protocols.

However, layered architectures have also some disadvantages [Garlan and Shaw 1993]. First, finding the right abstractions for the layers can be difficult, especially when creating standardized models. For example, there were some difficulties mapping existing protocols into the OSI model because they bridged several layers. Second, a closer coupling between layers could often improve performance. This is particularly important in the domain of resource-constrained sensor networks. For instance, no duplicate state information would be needed if layers shared their internal data such as information about neighboring nodes. This would not only reduce memory consumption but also reduce the processing and energy overhead for gathering this data. In addition, such a more closely interacting system could better deal with some properties of wireless transmission than a strictly layered architecture. For example, in wireless networks transmission failures are more common than in wired networks. However, if a packet is lost, TCP assumes network congestion and reduces its transfer rate. If it were informed about the cause for a loss – as with the Explicit Congestion Notification mechanism [Shakkottai et al. 2003] – better throughput could be achieved with wireless links.

Such interactions that couple logically separate layers more closely or even merge them are called cross-layer interactions. In particular, we define a cross-layer interaction as any form of interaction that is beyond the narrow functional interfaces of a strictly layered reference architecture (see Fig. 2.4(b)).

This definition is not limited to the layers of the network protocol stack but can be applied to other layered architectures as well. Furthermore, if the architecture does not consist of explicitly defined layers, interactions between other logically separate entities such as software components can be viewed as cross-layer interactions.

In this thesis we focus on cross-layer interactions that are intended by the developer to do some optimizations. We do not target unintended cross-layer effects like those occurring when using TCP over a lossy wireless channel [Kawadia and Kumar 2005].

There are several examples for cross-layer interactions described in the literature. For example, the MAC and routing layers often share some data like information about nodes in the neighborhood [Polastre et al. 2005a] or the local topology [van Hoesel et al. 2004]. Likewise, maps provided by the application can be used to improve the performance of geographic routing [Tian et al. 2003]. Another example can be found in query processing systems: The network and the application layers collaborate to aggregate data [Gehrke and Madden 2004]. Chapter 4 includes a description of cross-layer interactions in existing applications.

As already mentioned in the introduction, cross-layer interactions are frequently used in sensor networks. In fact, the domain of sensor networks is particularly suited for cross-layer interactions. First, there is usually just a single application running on each node, which allows for optimizations throughout all layers of the protocol stack. Second, in current operating systems the source code of this stack is available and – compared to other systems – more layers are implemented in software. For example, even the MAC layer is usually just another software layer and, therefore, modifications of the radio chip are often not necessary for cross-layer interactions. Finally, in TinyOS [Hill et al. 2000], for example, the traditional layered approach is substituted by component-based architectures that allow to perform cross-layer optimizations more easily [Levis et al. 2004a]. In such systems there is usually no strict separation between application and system components which further facilitates cross-layer interactions.

So far, however, cross-layer interactions have been most often used in an ad hoc fashion, i.e., without a structured mechanism. Therefore, their negative effect on modularity is greater than necessary. This leads to so-called spaghetti code, which is hard to understand and impossible to maintain. Because of these effects some skepticism about cross-layer interactions has emerged [Kawadia and Kumar 2005] – despite the obvious need for them in resource-constrained sensor networks. However, if cross-layer interactions were used with appropriate abstractions, many of their negative side effects could be alleviated. This approach is taken by most other cross-layer frameworks [Conti et al. 2004; Kumar et al. 2006; Su and Lim 2006] which focus on providing a structured mechanism to exchange data among layers (see Chapter 4). Such mechanisms help to better preserve modularity while still enabling data exchange. Furthermore, by addressing the causes of cross-layer interactions the system software can help to avoid some cross-layer interactions completely (see Chapters 5 and 6).

2.3. TinyCubus Project

This dissertation has been created within the TinyCubus project [Marrón et al. 2005a; Marrón et al. 2005b]. TinyCubus is a generic framework that supports the requirements of flexibility, adaptation, and reconfiguration of typical sensor network applications. This section gives a brief overview of the architecture of TinyCubus.

TinyCubus is implemented on top of TinyOS. It consists of three parts: the Tiny Data Management Framework, which supports the adaptation of components, the Tiny Configuration Engine, which allows for the exchange and reconfiguration of components at runtime, and the Tiny Cross-Layer Framework, which is described in detail in this dissertation.

2.3.1. Tiny Data Management Framework

The Tiny Data Management Framework is a set of system components that provide adaptation functionality. For each type of standard data management component such as replication, caching, hoarding, prefetching or aggregation, as well as each type of system component, such as time synchronization and broadcast algorithms, TinyCubus assumes that several implementations of each component type with the same interfaces exist. The Tiny Data Management Framework is then responsible for the selection of the correct implementation based on the current information available in the system.

The cube of Fig. 2.5, called “Cubus”, combines optimization parameters (O_1, O_2, $\ldots$), such as energy, communication latency and bandwidth; application requirements (A_1, A_2, $\ldots$), such as reliability or consistency level; and system parameters (S_1, S_2, $\ldots$), such as mobility or node density. For each component type, algorithms are statically classified in advance according to these three dimensions. For example, TAG [Madden et al. 2002] implements a tree-based routing algorithm used for aggregation in sensor networks that operates efficiently in static environments, but cannot be used

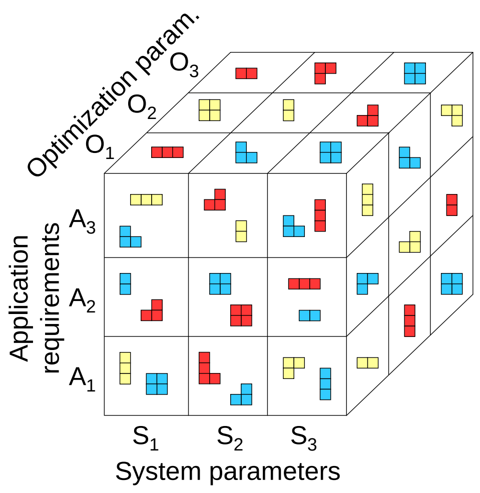


Figure 2.5.: Components in the Tiny Data Management Framework

effectively in highly mobile scenarios with strict reliability requirements. In such a setting, a flooding-based algorithm would probably do much better. Therefore, the component implementing the algorithm is tagged with the combination of parameters and requirements for which the algorithm is most efficient. The mapping of components to parameter values is performed off-line using experimental evaluations of each component in combination with the corresponding parameters. This way it is possible to know which components and/or groups of components perform best for a given parameter combination.

The Tiny Data Management Framework selects the best suited set of components based on current system parameters, application requirements, and optimization parameters. This adaptation has to be performed throughout the lifetime of the system and is a crucial part of the optimization process.

2.3.2. Tiny Configuration Engine

When new functionality such as a new processing or analysis function for sensed data is required by the application, it is necessary to install new components or swap functions. The Tiny Configuration Engine addresses this problem by distributing and installing code in the network. Its goal is to support the configuration of arbitrary components with the assistance of its two main parts: the topology manager and a runtime dynamic linking mechanism.

The topology manager is responsible for the self-configuration of the network and the assignment of specific functionality to each node. For this purpose it makes use of a generic role assignment mechanism [Römer et al. 2004]. Using the cross-layer framework's state repository it also publishes topology information that describes the neighborhood of sensor nodes, the status of communication links and the availability of components in other neighboring nodes. This and other cross-layer information can be used for the selection of more efficient routes for data and code dissemination as shown in [Marrón et al. 2005a].

The linking mechanism, FlexCup, allows for the reconfiguration of sensor nodes by providing the necessary bootstrapping code and the ability to load and install components on-the-fly. Using this approach the energy spent for code updates is largely reduced. In addition, it provides the flexibility needed for adaptation [Marrón et al. 2006a].

2.3.3. Tiny Cross-Layer Framework

The Tiny Cross-Layer Framework provides a generic interface to support the use of cross-layer interactions. It includes functionality to facilitate the exchange of data among components of different layers and among neighboring nodes, provides a virtual-memory abstraction, and offers a mechanism to meet user-defined lifetime goals. The Tiny Cross-Layer Framework is the focus of this thesis.

3. Design Overview

This chapter gives an overview of the cross-layer framework's design. It first presents the framework's functionality in some more detail and then describes properties of its design.

3.1. Functionality of the Cross-Layer Framework

As described in Chapter 1, the cross-layer framework pursues three ways to reduce the negative effects of cross-layer interactions and to avoid some of them completely. First, it provides a mechanism to exchange data among different layers and between neighboring nodes. Second, it includes a flash-based virtual memory system that tries to optimize the memory layout. Finally, it allows to meet a user-defined lifetime goal for the sensor nodes while taking into account their energy constraints.

Data exchange is one of the most commonly used techniques to implement cross-layer interactions. However, it increases the coupling between layers significantly. Therefore, the first part of the cross-layer framework, which is described in Chapter 4, tries to reduce this coupling. With this approach, which is called *TinyXXL*, the components exchange their data via the *TinyStateRepository*, a repository containing all cross-layer data. This way components do not have to interact directly to exchange data. *TinyXXL* optimizes the application to only include each data item once in limited memory and avoid – possibly energy-expensive – redundant data gathering. In addition, *TinyXXL* includes *Neidas*, an algorithm for neighborhood data sharing. This way not just the data from components on the same node but also the data from neighboring nodes can be accessed via the *TinyStateRepository*. *Neidas* tries to reduce the number of messages sent with "polite gossiping", which makes use of the broadcast nature of radio communication.

The second part of the framework, *ViMem*, deals with the memory constraints of sensor nodes by providing a virtual memory abstraction. Therefore, the memory constraints of sensor nodes are reduced and fewer cross-layer interactions are necessary. *ViMem* takes into account the special characteristics of sensor networks and the properties of flash memory, which is used as secondary storage. It uses traces from simulation to find out which variables are accessed more frequently than others and which ones are often used together. Based on this information it employs a heuristic

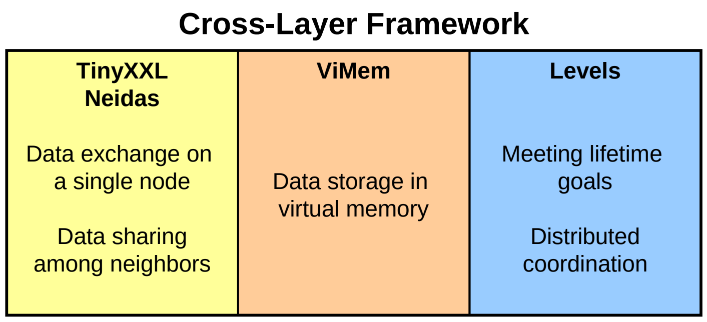


Figure 3.1.: Parts of the Cross-Layer Framework

to modify the memory layout in order to decrease the number of page faults. *ViMem* is described in Chapter 5.

The last part of the cross-layer framework, *Levels*, allows to meet a user-defined lifetime goal by reducing the functionality of the application if necessary. For example, a node could stop energy-expensive data sampling and just forward radio messages from other nodes to keep the network connected. Although the node does not provide its full functionality, it is still more useful this way than after failing completely. To achieve this, the developer has to specify optional functionality in the code and define a lifetime goal. The lifetime estimation at runtime is based on information from simulation tools that include detailed energy models of the sensor nodes. With *Levels* some of the cross-layer interactions caused by energy limitations can be avoided because the system itself takes care of ensuring that a node achieves the target lifetime. Furthermore, *Levels* includes an efficient distributed coordination algorithm that tries to keep overall application quality of the network roughly constant. *Levels* is described in Chapter 6.

Fig. 3.1 shows an overview of the three parts of the cross-layer framework. In the left column *TinyXXL* and *Neidas* form a comprehensive approach for data sharing. With this part of the framework we try to alleviate the negative side effects of cross-layer interactions. The center column shows *ViMem*, which addresses the memory constraints with its virtual memory system. In the right column *Levels* deals with energy limitations. It provides functionality to meet a user-defined lifetime goal and to coordinate nodes for constant application quality.

3.2. Common Design Properties

In the design of the cross-layer framework we considered specifically the resource limitations of sensor nodes. Therefore, two general principles are used to reduce resource consumption. First, the framework has been designed using a modular approach which avoids that unused functionality consumes resources. Second, as much processing as possible is done offline to reduce the runtime overhead.

Following a modular approach, the three parts of the framework can be either used individually or in combination. Using them individually allows to build a lean system where only those runtime components that are actually needed are included in the code. A similar approach is also pursued by TinyOS [Hill et al. 2000; Gay et al. 2003] where only those system components that are needed by the application become part of the compiled code image.

To shift processing offline, each of the three framework parts consists of compile-time tools in addition to the components of their runtime system. Whenever possible, these tools try to reduce the runtime overhead – including both processing and memory overhead.

The following subsections give a brief overview of the characteristics of the runtime system and the compile-time tools. A more detailed description of each of them can be found in Chapters 4, 5, and 6.

3.2.1. Runtime System

Parts of the runtime system are supplied with the framework and others are generated by the compile-time tools. These generated components are specific to the application at hand and include, for example, an optimized memory layout for virtual memory. Creating these components for each application allows to reduce RAM consumption since they can statically include most information needed at runtime. Therefore, it can be accessed from less constrained program memory instead of RAM.

Fig. 3.2 shows how the parts of the cross-layer framework interact at runtime. All three parts, i.e., the *TinyStateRepository* that stores *TinyXXL*'s cross-layer data, the virtual memory system *ViMem*, and *Levels*, the part dealing with energy consumption, can be accessed from the application and the system software. The figure also shows which hardware components are primarily used by which part of the framework: the *Tiny-StateRepository* uses radio communication to share data with neighboring nodes via *Neidas*, *ViMem* stores variables both in RAM and flash memory, and *Levels* estimates the remaining energy using the battery voltage sensor available on most sensor nodes. Besides the cross-layer framework, these hardware components may, of course, be accessed by other parts of the system, too.

Depending on the configuration, the *TinyStateRepository* can store its data either using the virtual memory abstraction of *ViMem* or in RAM. In order to allow for an accurate prediction of node lifetime, *Neidas* and *ViMem* use *Levels* to encapsulate the communication and flash memory code, respectively (see Chapter 6 for more details).

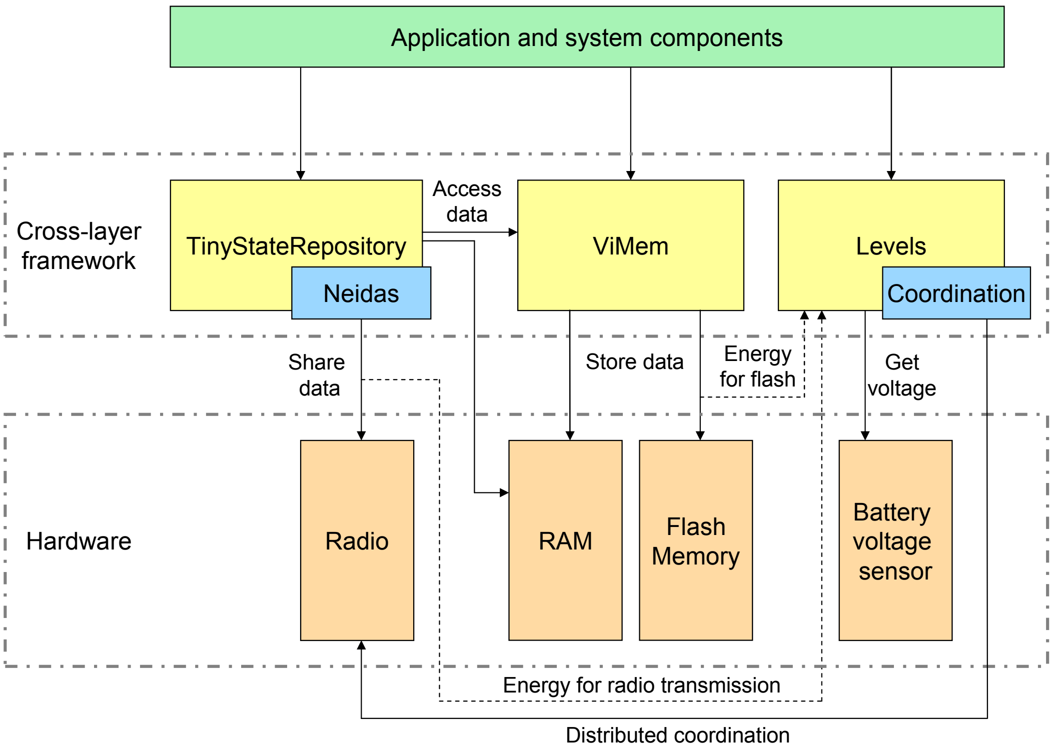


Figure 3.2.: Overview of the runtime system

3.2.2. Compile-Time Tools

For each of the parts of the cross-layer framework some extensions to nesC [Gay et al. 2003], the programming language used by TinyOS [Hill et al. 2000], have been defined. *TinyXXL* introduces data definitions and dependencies, *ViMem* allows to tag variables that are to be placed in virtual memory, and *Levels* introduces the energy levels abstraction. For each of the three parts a separate parser is called that processes these specific language extensions. Therefore, changes to existing tools such as the nesC compiler are minimal. In fact, no changes to the compiler are necessary. The only change that can be optionally applied exports some information from the nesC compiler to automatically ensure that *TinyXXL* variables accessed from time-critical functions are not placed in virtual memory.

After processing the input files the compile tools generate the code for the runtime system that is specific to the application. For *TinyXXL* they create the components of the *TinyStateRepository* and do the optimizations described in Chapter 4. Regarding *ViMem* each access to virtual memory is redirected to the runtime system and an optimized memory layout is computed. For *Levels* the compile tools insert information about the energy consumption of code blocks that are put into optional energy levels.

The compile-time tools have been integrated into the TinyOS build system. Therefore, with simple changes to the makefile an application can make use of the functionality of one of the cross-layer framework's parts. Their output is nesC code that is then processed by the standard nesC compiler.

Since *TinyXXL* and *Levels* require some extensions to the nesC language, their code cannot be processed by the nesC compiler before it has been modified by the framework's compile-time tools. However, because the implementations of both *ViMem* and *Levels* require some information from the nesC compiler, the order in which the three parts of the cross-layer framework are processed is important. In addition, some parts generate code that has to be included in the processing of the other ones. For example, *TinyXXL* can store its data in virtual memory and its neighborhood distribution algorithm makes use of energy levels. Therefore, the build process calls the compile-time tools in the following order. First, the *TinyXXL* data sharing code is processed. Its compile tool can deal with the language extensions of the other parts and does not require information from the nesC compiler. Second, *Levels*-specific code is processed. If *ViMem* is active, the *Levels* pre-compiler also processes the components in which *ViMem* uses energy levels for flash accesses – even if these components are not used by the application yet. Finally, the *ViMem* compile tool is called to create a memory layout and redirect virtual memory accesses.

For the application developers most of this process is transparent; they simply invoke the same build process like in a normal TinyOS application.

4. Cross-Layer Data Exchange

This chapter contains an analysis of cross-layer interactions in existing sensor network applications. Based on the specific instances found there it proposes a classification and identifies two types of cross-layer interactions – data sharing and parametrization – for which system support is especially promising.

Starting from these results the chapter describes the part of the cross-layer framework dealing with the exchange of data among both software components of a single node and neighboring nodes. Finally, it evaluates this part of the framework and gives a brief overview of related work.

4.1. Preliminaries

Although cross-layer interactions are often described in the literature [Melodia et al. 2005; Raisinghani and Iyer 2004; Srivastava and Motani 2005; Levis et al. 2004a], it is unclear what kinds of cross-layer interactions are actually used in real applications. To get a better understanding of this problem – and of cross-layer interactions in general – we analyzed several real-world applications and created a classification of their cross-layer interactions.

From this classification we then selected two types of cross-layer interactions that are not adequately supported by current programming languages: parametrization and data sharing. These classes of cross-layer interactions, which we subsume under the term "data exchange", are often used in practice for important optimizations. First, parametrization is essential to tailor system components to the specific requirements of an application, which allows for energy-efficient operation. Secondly, because of data sharing there is no need to keep redundant data in stringently constrained RAM and to acquire it twice with possibly high energy costs (e.g., for sending messages). Furthermore, they also help to deal with the special properties of radio communication.

In current programming languages such as nesC [Gay et al. 2003] these interactions are often implemented with function calls. As we show in Section 4.3, such an approach increases coupling unnecessarily and creates considerable overhead for the developers, especially when the application evolves. In addition, it hinders component reuse because the components forming the application have to be optimized by hand in order to prevent separate components from providing the same data twice. Therefore, we have created *TinyXXL* ("**Ex**change of **Cross**-**L**ayer Data for **Tiny**OS") that provides programming language support for data exchange [Lachenmann et al. 2006]. *TinyXXL* is composed of two parts: a compile-time and a runtime component. For compile-time support of data exchange, we extended the nesC programming language to include abstractions for data definition and exchange. At runtime this language extension is complemented by the *TinyStateRepository* that stores all cross-layer data and provides efficient access to it.

With *TinyXXL* and the *TinyStateRepository* we pursue the goal of creating language and system support for highly optimized applications while fostering component reuse and independent software development. First, we try to decrease the effort of application developers when exchanging data and to decouple interacting components. Secondly, our approach automatically optimizes applications at compile-time by removing redundant data provision code and selecting a data provider that meets the non-functional requirements of the data users best. Therefore, arbitrary reusable components can be combined to form an optimized application without any data redundancies. Thirdly, since most of the checks are performed at compile-time, there is only little runtime overhead associated with the *TinyStateRepository*. Finally, this part of our framework does not only support data exchange among components of a single node but also integrates an efficient algorithm for neighborhood data sharing.

This algorithm, *Neidas*, efficiently requests and sends neighborhood data by taking advantage of the broadcast nature of radio transmission [Lachenmann et al. 2007c]: it suppresses messages if, for example, several neighbors have already sent the same request. In addition, by periodically repeating packets the algorithm increases the probability that neighboring nodes actually receive the content.

4.2. Application Analysis

Although there are no traditional layers in component-based applications like those based on TinyOS, the components forming an application can be classified at a certain level of abstraction (e.g., hardware abstraction level or application level). These levels of abstractions can be regarded as layers. The concrete form of these conceptual layers is part of the design decisions taken by the application developer. In such architectures cross-layer interactions are implicitly allowed since there is no restriction as to which component can be used [Levis et al. 2004a]. With this view cross-layer interactions are not limited to the network protocol stack but can occur between any system or application components.

As described in Chapter 2, in this thesis we consider cross-layer interactions to be all interactions among components of logically separate layers that are beyond the use of narrow functional interfaces of strictly layered reference architectures to, for example, send and receive messages. Such interactions can skip layers or couple them more tightly.

4.2.1. Selected Applications

In order to provide a classification of cross-layer interactions that are actually used in real-world applications, we have analyzed existing sensor network applications and system components available in the TinyOS CVS repository [TinyOS] and those developed by our research group. Since TinyOS [Hill et al. 2000] is – by far – the most frequently used operating system for sensor networks, we focus on applications built on top of this system using the nesC programming language [Gay et al. 2003]. In nesC interactions based on the use of specific interfaces can form an arbitrary dependency graph. For instance, it is possible for a high-level application component to directly use a low-level hardware component.

We have analyzed three nontrivial applications, namely TinyDB, AcousticLocalization, and Sense-R-Us, which contain between 11,000 and 30,000 lines of code. TinyDB [Madden et al. 2005] provides query processing capabilities with an SQL-like query language and offers a generic solution for data gathering. AcousticLocalization can be used to determine the geographic position of nodes in a sensor network by taking advantage of the difference in the speed of radio waves and sound [Sallai et al. 2004]. It represents an example of a sensor network application that does more complex data processing in the network than simply aggregating the sensor readings. Finally, Sense-R-Us [Minder et al. 2005] is an application developed by our research group that uses a sensor network to provide functionality typically found in smart environments. Each user carries a sensor node that can communicate with stationary nodes placed in offices and meeting rooms. The application is then able to determine the position of research assistants, detect the location and duration of meetings, etc.

To put our classification on a broader basis, we have also analyzed the requirements and design of the Sustainable Bridges and CarTALK 2000 applications, which are the motivating examples for the TinyCubus project [Marrón et al. 2005b]. Sustainable Bridges [Marrón et al. 2005c] is an application for long-term structural health monitoring of bridges. Using different kinds of sensors it tries to detect small defects before larger damage can occur. Since batteries can only be replaced during regular bridge inspections, energy efficiency is very important for this application. CarTALK 2000 [Reichardt et al. 2002], in contrast, has the goal of creating a cooperative driver assistance system that provides an ad-hoc warning system for traffic jams, accidents, and lane or highway merging. Since sensors are integrated into cars, they move relative to each other and, therefore, algorithms that are able to cope with mobile sensors are needed to accurately process data.

As this description shows, these applications represent a wide range of sensor network applications with different properties.

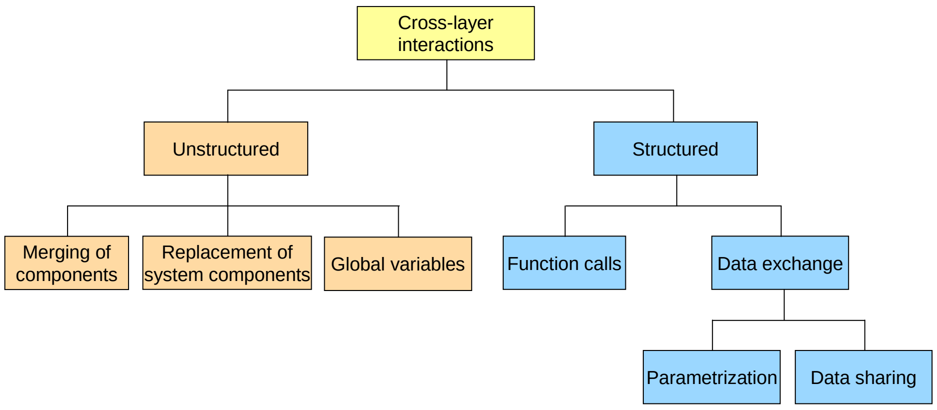


Figure 4.1.: Overview of different forms of cross-layer optimizations

4.2.2. Forms of Cross-Layer Interactions

Fig. 4.1 shows the taxonomy of cross-layer interactions we have created by analyzing the aforementioned applications [Lachenmann et al. 2005]. In the figure we distinguish between unstructured and structured forms of interaction. The unstructured interactions are located on the left-hand side and represent cross-layer interactions that interfere most with modular software development and that tend to make the programming and maintainability of code significantly harder. The first one, merging components, deals with combining components that are logically separate into a single one. The second type of interaction, the replacement of system components, can be found in situations where an application replaces certain system components with application-specific code that uses the same interfaces, but changes the default behavior of the system. Finally, the third one covers the use of global variables in order to facilitate the sharing of data between components.

The right-hand side of Fig. 4.1 depicts the class of cross-layer interactions that can be found in better structured applications. The first type deals with the use of interfaces for the execution of callbacks or functions in other layers in a controlled way. This is the way applications directly access hardware components and low-level components execute application-specific code. The second type, data exchange, is a structured alternative to global variables. In our view, there are two possibilities: either allowing the parametrization of components, or providing capabilities related to the controlled sharing of data.

It should be noted, however, that function calls and data exchange are not mutually exclusive. For example, data sharing can be implemented by using function calls that return the appropriate data to the caller. In fact, using languages like nesC this is the only way to implement data exchange.

The following paragraphs give a more detailed description of each kind of interaction using examples from the applications we have analyzed.

Merging of Components

The combination of components into a single one is normally used to allow for easier interaction among them. This kind of interaction does not require the separation of layers at all. However, it contradicts the principles of good software engineering since it does not allow for the easy and independent development, exchange, and reuse of components.

Merging components can often be avoided by creating separate components that co-operate by using either function calls or some form of data exchange. Therefore, there is usually a better alternative to simply combining code. Nevertheless, this simple form of cross-layer interaction is often seen in practice. For example, in TinyDB an application-layer component includes parts of the routing code because it needs to determine which node should receive aggregated results, for example. Applying this optimization helps to save both energy and main memory.

Replacement of System Components

Another form of unstructured cross-layer interaction is the replacement of system components. This is normally used to add specific functionality of the application components into lower layers that otherwise do not offer it. The problem with this kind of interaction is that side-effects on other system components and non-application components are hard to determine. Furthermore, if a new version of the component becomes available and implements additional functionality, the application that has replaced it does not automatically benefit from it. This makes the maintenance of the application harder since these modifications will have to be updated manually.

Therefore, a much better alternative is – if possible – to include additional application-specific components with similar functionality or to use callbacks in the real system components that implement the desired functionality. Just like merging of components, this type of interaction is also very common in applications. For example, TinyDB replaces SimpleTimeM with its own implementation that makes the component's accuracy parameterizable. If another component included in the application relies on the predefined accuracy value, the use of this modified component can lead to timing bugs which are hard to find. Likewise, AcousticLocalization replaces several TinyOS components that are related to sensor value reading and, again, timing. Compared to the standard components, the sensor access components add support to read a continuous series of samples and the timer component has been modified to provide an interface that allows for more fine-grained accuracy. The reasons for these changes are either hardware limitations (e.g., a small number of hardware timers) or inadequate abstractions provided by the original components. Finding the right level of abstraction in such generic components is a common problem that makes the creation of layered architectures difficult [Garlan and Shaw 1993].

Two more disadvantages become apparent with these components. First, they are only targeted to Mica2 nodes, which makes porting and simulating the whole application difficult. Second, the application-specific timer component has not been updated when the standard TinyOS component evolved. So it does not include support for power management, for example, as it is the case in more recent releases of TinyOS.

As a final example, in CarTALK 2000 the MAC layer has been developed specifically for this application and is used by all network protocols. Therefore, it replaces a standard MAC layer protocol, which might be expected by other protocols or applications on the nodes, with an application-specific one.

Global Variables

This form of interaction uses variables that are globally accessible from several components to exchange data. The main problem here is that the dependencies of software components that read or write these variables are not specified. Thus it remains unclear whether the data is actually needed, whether the variables are filled with values at all, or whether there are several components writing potentially inconsistent data.

In TinyOS-based sensor networks nesC's component-oriented programming model discourages the use of global variables although it is still possible to employ them. Instead, data is usually defined local to one component with access functions that can be used by other components.

In our sample TinyOS applications global variables are used in a limited way. In these applications only the node's network address and the application's group ID used for radio communication are stored globally. However, these variables are usually treated like constants. The only reason why they are defined as variables is to allow applications to dynamically set them at runtime after installing a new code image, for example.

Function Calls

Function calls are well-supported by nesC using commands and events that are defined in interfaces. They are needed not only for cross-layer interactions but also for both interactions of components within layers and calls to the functional interface of directly adjacent layers. With cross-layer interactions, however, function calls are needed for the use of specific functionality in lower layers or for the integration of high-level code in lower layers via predefined callbacks. In sensor networks, where there is no clear separation between system and application components, function calls can be made from any one component to any other simply by wiring their interfaces together.

As an example, Sense-R-Us calls low-level functions to turn on and off sensors as they are needed, and AcousticLocalization periodically turns off the radio module.

This is also done in the Sustainable Bridges application. Obviously, the reason for this behavior is the energy limitation of the nodes. Another example can be found with the components that acquire network neighborhood information for TinyDB and Sense-R-Us. They intercept all messages received by the node and store information about the senders. Thus they avoid energy-expensive communication for their own beacon messages. Finally, low-level networking components define a callback used by Sustainable Bridges and TinyDB to insert a time stamp into the message at the time it is being sent. This guarantees better accuracy for time synchronization. However, function calls increase the coupling between components because there is a direct connection between the interacting modules. By using generic aliases for the components being wired (e.g., "MAC_Component" or "Routing_Component") instead of the actual components' names this coupling could be partially reduced.

Although function calls can be used to implement parametrization and data sharing, we consider these two classes to be separate, data-centric forms of cross-layer interactions: For exchanging data a function call would unnecessarily increase coupling since the source of the data is most often not important. However, in nesC-based applications these forms of interaction have to be implemented as function calls because this programming language does not have separate abstractions for data exchange.

Parametrization

Parametrization is a cross-layer interaction that has to do with the explicit exchange of data. It is normally used to adapt the behavior of components using well-defined switches that change the functionality, execution path, etc. In contrast to data sharing, parameters target exactly one component and modify its behavior based on the requirements of the application.

Parametrization of system components in TinyOS-based sensor networks is possible because there is usually only one application actively running on the system at a given time. The components of this application cooperate and are usually tailored to fulfill the application's goals.

There are several examples of parametrization in the sensor network applications we have studied. Their most important characteristic is the fact that there is usually only one component that offers an interface for parametrization, and a set of components that use it to provide values. For example, the TinyOS MAC component, which is based on the B-MAC protocol [Polastre et al. 2004], has an interface to set the radio frequency, change the length of the preamble, and turn acknowledgments on or off. Another example for parametrization is the routing component in TinyDB. Its update interval is adjusted by an application-level component based on the data sampling rate of the queries currently being executed. Similarly, in Sustainable Bridges the routing algorithm is parameterizable. In addition, it makes use of parametrization itself by dynamically adjusting the transmission power of the radio. The very same

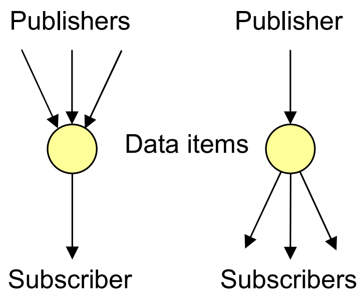


Figure 4.2.: Parametrization (left) vs. data sharing (right)

example of cross-layer interactions can also be found in CarTALK 2000. Adjusting the transmission power is a special characteristic of wireless networks that can help to reduce energy consumption and collisions.

If we talk about this cross-layer interaction in terms of a publish/subscribe system, parametrization implies the presence of a variable or a set of variables with only one subscriber (the component it parametrizes), and a series of publishers which modify its value (see the left-hand side of Fig. 4.2).

Data Sharing

Finally, data sharing involves the use of one or more variables in the program that are shared among components. This information is not related to parameters and can be stored, modified and computed dynamically. Unlike global variables the dependencies on the data are specified more clearly here.

Normally, data sharing is implemented using function calls that introduce tight coupling, and most of the time developers try to optimize it by hand to avoid duplicate data when reusing components.

In our analysis of the applications we did not encounter cases where several publishers provided a single piece of data to several subscribers. We attribute this to the fact that the applications are highly optimized by hand and do not include redundant code. Therefore, using the same model as for parametrization, data sharing can be considered as a form of cross-layer interaction where a piece of data has only one publisher, that is, the component that provides this piece of information to other components, and a series of subscribers that are interested in knowing it (see the right-hand side of Fig. 4.2).

There are many examples of such cross-layer interactions, since there is a large potential for memory and energy savings. Some of them deal with hardware configuration data or information about current queries in Sense-R-Us. In addition, in this application information about the network neighborhood is shared between communication components and an application component that uses this data to determine the cur-

Table 4.1.: Classification of unstructured cross-layer interactions

Merging of components	**Replacement of system components**	**Global variables**
Applications includes routing code (TDB)	Application-specific MAC-layer (CT)	Local address (all TinyOS applications)
	Time components replaced (TDB, LOC)	Radio group ID (all TinyOS applications)
	ADC components replaced (LOC)	

rent position of the node. Likewise, in TinyDB the routing components share much of their internal data with the application components (e.g., the routing parent, the hop count to the root, the occupancy of the send queue, and the set of neighbors). This data can be accessed via queries.

Even more examples can be found in Sustainable Bridges and CarTALK 2000. In Sustainable Bridges the routing metric uses cross-layer data about the quality of radio links and the energy available. This way it reduces energy consumption and deals with peculiarities of radio communication. Sustainable Bridges's role assignment algorithm requires both local and neighborhood data from several layers to evaluate the role specification. The role data, in contrast, is used by the code distribution algorithm to efficiently disseminate code updates in the network. Again, this helps to reduce energy consumption.

In CarTALK 2000 the routing and aggregation algorithms use cross-layer data in the form of a road map, the node's own position, and that of its neighbors to efficiently route and aggregate messages. In addition, the adaptation algorithm uses cross-layer data of several layers to make its adaptation decisions.

4.2.3. Summary

Table 4.1 and Table 4.2 classify the cross-layer interactions described above into our taxonomy. All the cross-layer interactions identified in the applications can be assigned to the categories of our classification. It should be noted, however, that – like every design decision – finding cross-layer interactions is to some extent subjective. Therefore, another analysis might identify different cross-layer interactions in the applications.

In the tables, "TDB" is short for TinyDB, "LOC" for AcousticLocalization, "SRU" for Sense-R-Us, "SB" for Sustainable Bridges, and "CT" for CarTALK 2000.

Most of the cross-layer interactions used in the applications are structured, i.e., function calls, parametrization, or data sharing. As described above, the unstructured forms of interaction lead to bad software design, hinder modularity, and limit reusability. Better alternatives are given by the more structured forms of interaction that are often related to the unstructured ones. For example, data exchange is the structured

Table 4.2.: Classification of structured cross-layer interactions

Function calls	**Parametrization**	**Data sharing**
Turn off hardware devices (SB, TDB, SRU)	MAC layer parameterizable (all TinyOS applications)	Routing uses link quality, energy (SB)
Time synchronization when sending (SB, TDB)	Routing parameterizable (SB, TDB)	Role assignment uses data (SB)
Application intercepts packets (TDB, SRU)	Adjust transmission power (SB, CT)	Code distribution uses roles (SB)
		Routing uses road map, positions (CT)
		Aggregation uses road map, positions (CT)
		Adaptation uses cross-layer data (CT)
		Configuration data (SRU)
		Current queries (SRU)
		Neighborhood information (SRU)
		Internal routing data (TDB)

version of global variables. Likewise, function calls can often substitute merging of components and replacement of system components.

The tables show that a large number of cross-layer interactions deals with parametrization and data sharing. As already described above, these forms of interactions are not well-supported by current programming languages and are often implemented with function calls. In addition, in our experience these forms of cross-layer interactions have a large potential for optimizations. Therefore, based on these findings, we have developed *TinyXXL*, which is described in the following section.

Furthermore, as the description in the previous subsection illustrates, many cross-layer interactions are caused by energy limitations, memory constraints, and special properties of wireless communication. These are the properties that we specifically address with our cross-layer framework. However, some interactions are applied because of different reasons like, for example, other hardware limitations or inadequate abstractions of the system components. Nevertheless, in some cases these interactions can also be addressed with our framework.

4.3. Data Exchange on a Single Node

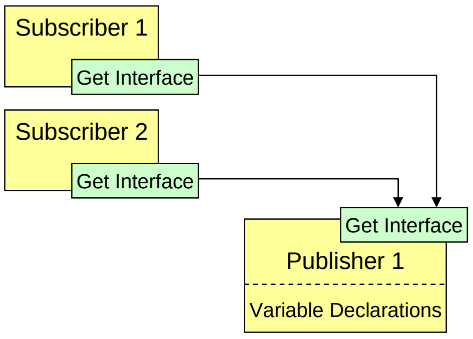
Current programming languages do not provide explicit and adequate support for data exchange. For example, nesC only allows the implementation of data exchange using function calls, which unnecessarily increases coupling and which can lead to a significant development overhead as well as possibly unoptimized applications. First, the developer has to create an interface, implement this interface within a component, and “wire” all users of the piece of data to this component. Usually, this direct interaction is not needed since the components are only interested in the data – independent from its source. Secondly, if two components provide the same piece of data, this data is stored and acquired twice, which might require energy-intensive operations such as sending messages. For example, if both MAC layer, routing, and application-level components maintain neighbor tables, they allocate limited memory space for redundant data. In addition, they waste processing time and energy, if they send beacon messages to update these tables.

Manually optimizing the components that form an application increases the development efforts significantly. With sensor network applications becoming more and more complex, inefficiencies due to duplicate data are hard to detect and even more so to fix. For example, TinyDB, consists of almost 30,000 lines of code grouped in 176 components. Therefore, when developing such a complex application, it is difficult to detect components with duplicate data. It is even more difficult to optimize the application by removing data gathering code so that the same data is not stored and acquired twice. Often this code is not isolated in a special function but interwoven with other functionality needed for the component to work properly. In addition, as we describe in the following paragraphs, such modifications hinder reuse and independent development of components, two approaches often used to significantly decrease the development costs of software.

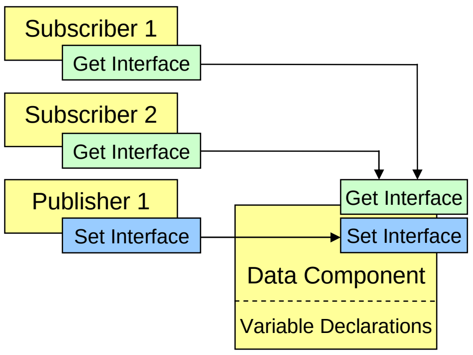
Basically, there are two common solutions to properly implement data exchange with nesC: The first one stores the data in the component sharing its data whereas the second solution uses a separate component to store the data that is wired to both its providers and subscribers.

Storing the data within the providing component (see Fig. 4.3(a)) is most often used in existing applications. However, this solution introduces tight coupling between components accessing and providing a given piece of data, which hinders maintainability. If a component providing some data is to be replaced, not only the wirings of its functional interface have to be adjusted but also those of the components that use its data. These wirings (the arrows in the figure) can be distributed across all application and system components since in nesC any component may use any other one. For example, an application component that does not send any radio messages itself might access the network neighborhood information of the routing component.

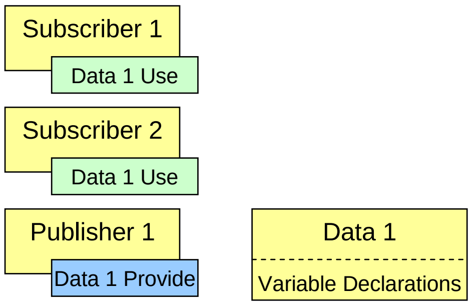
If data is stored in a separate component that has been exclusively created for data



(a) Data stored in publisher



(b) Data stored separately



(c) Data declared with *TinyXXL*

Figure 4.3.: Possibilities to declare shared data with nesC and *TinyXXL*

storage (such as in Fig. 4.3(b)), components accessing and providing some piece of data are decoupled; so the aforementioned maintainability problems do not appear with this solution. However, the compiler cannot guarantee automatically that there is a component in the system providing the data. In addition, to avoid duplicate data provision, publisher components have to check whether or not they have to acquire the data. This check is difficult to implement without runtime overhead, especially if non-functional requirements have to be considered. For example, such requirements could be a certain accuracy level or an update frequency needed by the data user.

We address these issues by extending nesC with *TinyXXL* in the following way:

- *TinyXXL* decouples the components providing and using data (see Fig. 4.3(c)) by automatically creating wirings between them and by using a publish/subscribe scheme, which eases the process of data exchange.

- For shared data *TinyXXL* ensures that there is only a single component providing each data item. In contrast, with parametrization several components can provide values to modify the behavior of a specific one, such as the MAC layer component.

- *TinyXXL* adds capabilities for the specification of non-functional properties of data providers so that the system can select the one component that meets the requirements of data users best.

- *TinyXXL* provides efficient automatic notifications of subscribers after changes to the data.

- *TinyXXL* offers optimization capabilities that remove the data gathering code from all but one provider of a single kind of data. This way, no processing time and energy is spent for acquiring redundant data. Thus it is possible to develop optimized applications from reusable components without manual intervention. For example, both a generic MAC layer and a routing component can provide information about the quality of network links. With *TinyXXL* only one of them gathers this data; thus the size of allocated memory and – possibly – energy consumption are reduced.

4.3.1. TinyXXL Language Description

The changes to nesC needed to achieve the benefits mentioned above are relatively simple. Our additions and modifications include the ability to declare data definition files, specify data dependencies, specify "**ifproviding**" blocks for data publishers, and use virtual data items. For reference, Appendix A summarizes the changes to the nesC grammar.

```
xldata NeighborData(cost_type buildCost(<)) {
  NeighborNode neighborTbl[ROUTE_TABLE_SIZE];
  int8_t neighborCount;
}
```

Figure 4.4.: Declaration of shared data with *TinyXXL*

Data Definition

Within *TinyXXL* data is defined in a separate file similar to the way interfaces are specified in nesC. In such a file the developer groups all the data items that logically belong together. For example, an array with information about the neighboring nodes is declared in the same file as a counter for the number of elements in it (see Fig. 4.4).

The syntax of the definition of individual data items resembles the declaration of variables. Unlike interfaces, which can be implemented by several components, there is only a single instance of a data file. This way the data can be identified by its unique name. As the example above shows, it is possible to declare parameters for non-functional requirements (in this case "buildCost"), which are used by the system to select a publisher component that meets the requirements of the subscribers. The "less than" sign in the example expresses that the publisher with the smaller values for this parameter should be preferred if several ones fulfill these requirements. This structure provides hints to *TinyXXL* regarding possible optimization strategies. Depending on the kind of data, other requirements (e.g., concerning the update frequency of a data item or its accuracy) can be added.

In the case of parametrization several components may set the values influencing a single parameter. Therefore, there is no need to select a publisher meeting non-functional requirements; such requirements have to be defined only for shared data.

Specification of Data Dependencies

Components exchanging data declare this property in their header similar to the interfaces provided. However, in contrast to interfaces there is no need to create wirings for data dependencies because they are automatically resolved by the *TinyXXL* compiler. If data definition files specify non-functional properties, components providing data have to give values for those here. Components subscribing to the data may specify an arbitrary condition that uses a data item's non-functional requirements. This condition has to be met by the publisher component. For example, a data subscriber can specify that the cost for acquiring some data may not exceed a given limit and that the accuracy has to be better than some threshold.

Fig. 4.5 shows an example for a component publishing some data and subscribing to a parameter. From the developers' point of view data is accessed like global variables.

```
module MultiHopRouter {
  provides {
    xldata NeighborData(COST_PERIODIC_MSG);
    ...
  }
  uses {
    xlparam RoutingParam;
    interface ReceiveMsg;
    ...
  }
}
implementation {
  event TOS_Msg* ReceiveMsg.receive(TOS_Msg* Msg) {
    ifproviding (NeighborData) {
      ...
      NeighborData.neighborTbl[iNbr].address
        = pRP->source;
      NeighborData.neighborTbl[iNbr].refresh
        = NBR_MOST_RECENT;
      ...
    }
  }
  event void RoutingParam.changed() {
    call Timer.stop();
    call Timer.start(TIMER_REPEAT,
      RoutingParam.updateInterval * 1024L);
  }
}
```

Figure 4.5.: Provision of data and use of parameters with *TinyXXL*

The difference is, however, that each variable name has to be preceded by the name of the data definition it is contained in (e.g., "NeighborData.neighborTbl"). Again, this is analogous to the use of interfaces in nesC. Another difference compared to global variables is that data accesses of publishers have to be included in an "**ifproviding**" block (see below).

If problems from concurrency can arise, developers have to enclose accesses to shared data in standard nesC "**atomic**" statements (not shown in the code example). Therefore, they can use the constructs that they are already familiar with from nesC.

Neither publishers nor subscribers can access the data via pointers to guarantee that only components declaring correctly their dependencies are able to access the data. Our experience after modifying and creating several nontrivial applications using *TinyXXL* shows that this limitation does not severely restrict the developer.

When a component subscribes to some data, the developers have to implement a special function named "changed" (see Fig. 4.5). This is a notification function called after data has been modified. If this functionality is not needed, the compiler removes this code. As we describe in Section 4.3.3, data subscriptions themselves cannot be changed at runtime but are statically created at compile-time.

"ifproviding" Blocks for Data Publishers

For components publishing some data we added the "**ifproviding**" blocks as another language construct. All the code related to data provision has to be included in such a block (see Fig. 4.5). This is necessary for two reasons. First, if the component does not have to provide the data because another one supplies the same data, the code inside this block is removed by the compiler. So there are no unnecessary processing steps to provide the same data twice and no overhead at runtime to check if the component has to acquire the data. Secondly, at the end of such a block subscribers are automatically notified of the change. These notifications cannot be accidentally omitted by the developer and this solution offers higher efficiency than notifications after each variable assignment. There is no need for "**ifproviding**" blocks if a component just subscribes to some data; it may access the data anywhere in the code.

Obviously, code that is needed to fulfill a publisher's functional purpose cannot be encapsulated in an "**ifproviding**" block. Otherwise, the component could not work properly if another component publishes this data and the code within the "**ifproviding**" block is removed by the compiler. In this case the component also has to specify a dependency on the data as a subscriber so that it can access the data outside the "**ifproviding**" blocks. For example, a routing component that makes its internal neighborhood information available to other components also has to subscribe to this data if it uses it for routing.

Virtual Data Items

Besides data stored in RAM, we have added support for dynamically generated data with virtual data items. Virtual data items declare how the data can be computed dynamically from some other data already present. For subscribers, this is completely transparent; they cannot tell whether data is stored in RAM or generated on-the-fly.

Using virtual data items, operators known from databases can be implemented for cross-layer data. This functionality can include projections from several data definitions into a single one and aggregation functions that perform some computation on the data. For example, Fig. 4.6 shows a virtual data item that aggregates the number of neighboring nodes by counting the elements in some other data structure. Similarly, if there is no data publisher for the kind of data needed by a subscriber in the system, virtual data can be used to convert some other data to the required format. For example, a component just interested in neighborhood information does not want to process complex routing information, although the requested data could be inferred from that. Therefore, a virtual data item can distill this information from the more

```
xlvirtual NeighborCountAggregator {
  provides xldata NeighborCount();
  uses xldata RoutingData();
}
implementation {
  xldata void NeighborCount.count(uint8_t* result) {
    uint8_t i;
    *result = 0;
    for (i=0; i<MAX_ELEMENT_COUNT; i++) {
      if (RoutingData.neighbors[i].flag != EMPTY)
        (*result)++;
    }
  }
}
```

Figure 4.6.: Sample virtual data item that aggregates the number of network neighbors

complex internal data of the routing component without increasing allocated memory. These conversion capabilities can also be used with evolving data definitions, as new versions of a component are developed. Here components still expecting the old data format can use virtual data instead of the actual representation in RAM. The decisions on whether to store some data in RAM or to provide it using virtual data is made by the *TinyXXL* compiler based on the publishers and subscribers available.

There are several advantages of virtual data items. First, it is possible to provide additional data besides just the internal representation of the publisher components. This way data that is not directly provided by the components in the system can still be used by subscribers. Secondly, using virtual data reduces the amount of data that is stored in limited RAM and does not impose the overhead of acquiring similar data twice. Thirdly, instead of requiring every publisher or subscriber to convert the data, the system provides the data already in an immediately usable format.

To implement a virtual data item, the dependencies on other data have to be specified just like with data accesses. Then for each variable declared in the represented data a function like the "NeighborCount.count" function in Fig. 4.6 has to be provided. Obviously, this function can incur some processing overhead. However, this processing overhead would also exist with conversions that are implemented in pure nesC code and is often less than acquiring equivalent data twice.

4.3.2. Impact on the Life Cycle of Applications

Using *TinyXXL* influences the whole life cycle of sensor network applications, including design, implementation, and operation. In the design phase the developer can select reusable components of which the application is composed without making sacrifices regarding cross-layer optimizations. In addition, despite the use of cross-layer interactions, the modularity of the application is preserved and components are decoupled. Thus, when the application evolves, components can be exchanged more easily.

In the implementation phase the developer does not have to manually optimize data exchange, e.g., by ensuring that no redundant data is gathered and stored. This is something that is already done by *TinyXXL*. Therefore, the implementation effort is largely reduced (see Section 4.5.1).

During the operation phase of a sensor network application *TinyXXL* helps in reducing resource consumption by automatically performing cross-layer data exchange. Although the developer does not have to deal with optimizations, the performance of an application built from reusable components is comparable to a manually optimized one, since no redundant data is acquired and stored.

4.3.3. TinyXXL Compiler

The *TinyXXL* compiler is a pre-compiler that outputs pure nesC code. It has been implemented in Java using JavaCC as a parser generator. From the data definitions and the data dependencies the *TinyXXL* compiler generates the files that implement the *TinyStateRepository* (see Section 4.3.4). It ensures that only components declaring their dependencies can access the data in the specified way. The compiler resolves non-functional requirements on publishers by adding preprocessor directives that select the data provider satisfying the requirements best.

Since the *TinyXXL* compiler resolves all data dependencies at compile-time, it is not possible that components subscribe to some data dynamically. We selected this approach for two reasons. First, our analysis of existing sensor network applications (see Section 4.2) did not show much need for dynamic subscriptions. Secondly, this approach can be implemented without any RAM overhead and notifications can be realized more efficiently because there is no list of current subscribers to check.

The *TinyXXL* compiler translates all data accesses into function calls to the *TinyStateRepository*. This way it ensures that the data cannot be accessed via pointers. Like almost all nesC functions these function calls are later inlined by the nesC compiler so that the compiled code closely resembles direct variable accesses. Furthermore, each "**ifproviding**" block is translated into a regular if-statement that can be evaluated at compile-time. Thus none of the *TinyXXL* concepts imposes the runtime overhead of a function call.

4.3.4. Runtime Support for Data Exchange

At runtime the *TinyStateRepository* provides support for cross-layer data exchange. It consists of a set of components generated by the *TinyXXL* compiler and stores the data and parameters specified using *TinyXXL*. For each data and parameter declaration the

TinyXXL compiler creates a nesC component that declares data as variables within this component. This file also contains access functions to get and set the values of variables in a controlled way. So the code generated is similar to the example shown in Fig. 4.3(b). The subscribing components are automatically wired to the get interface and one of the publishers is selected to provide this data, which is wired to the set interface. In this example “Publisher 2” does not need to provide the shared data. Therefore, the *TinyXXL* compiler removes the code related to data provision from this component. The *TinyStateRepository* then automatically notifies all subscribers after the publisher has finished writing data.

In the *TinyStateRepository* the system keeps information about the name of the data item, its type of cross-layer interaction (parametrization or data sharing), a list of publishers of each data item, a list of subscribers, its data type, and the value of the shared data or parameter. Only the data values themselves are kept in RAM; all other information is translated at compile time and implicitly stored in the code image of the application.

The solution described so far requires the compiler to do most of the work like checking data dependencies. In addition, with its global view of the application the compiler is able to remove code that is not needed. However, such compile-time optimizations are not possible when using adaptation within the TinyCubus framework (see Section 2.3). If such adaptation capabilities are to be supported, there is no longer a global view at compile-time since binary components can be linked to the code image individually. However, FlexCup [Marrón et al. 2006a], the linker on the sensor node, can be modified to do most of the optimizations during adaptation, which are performed by the compiler in the static case. For example, just like functional dependencies it can also check if all data dependencies are fulfilled and use the non-functional requirements to select a publisher that meets the requirements of the subscribing components best. By directly changing parts of the program code (i.e., the values of some constants) the linker can perform these changes without increasing RAM consumption and with only little runtime overhead for accesses to the *TinyStateRepository* (see Section 4.5.1). The only difference is that, since TinyCubus does not compile any code on the sensor nodes, the code within the “**ifproviding**” blocks is not removed but simply not executed if it is not needed.

4.3.5. Advantages

The use of *TinyXXL* exhibits several advantages compared to pure nesC solutions.

- *TinyXXL* ensures the modularity of applications by having components explicitly declare their dependencies on shared data and parameters. Therefore, it is not necessary to directly wire components accessing some data to those that provide it.

- The components exchanging data are decoupled from each other and, when the application evolves, can be replaced independently.
- The *TinyXXL* compiler automatically reduces resource consumption by removing code that acquires redundant data and by selecting a single publisher component that fulfills the requirements of the subscribers with the least costs.
- Since redundant data is automatically removed from applications, components, which – in addition to their actual functional purpose – provide some of their data to other components, can be developed without wasting memory and energy. These components can then be reused in new applications, regardless of other components that possibly provide the same types of data.
- *TinyXXL*'s tight integration in a programming language allows us to perform checks at compile time without any runtime overhead and with only little overhead if support for dynamic adaptation is needed. For example, the *TinyXXL* compiler ensures important properties such as type safety as well as access control so that only components declaring their dependencies correctly on some data or parameter can access it.
- The integration in the programming language allows for an efficient publish/subscribe mechanism without any RAM overhead and with automatic notifications of changes.

4.4. Neighborhood Data Sharing

Many protocols and applications do not only require data from other layers on the same node but also data from neighboring nodes. This data is either required to provide the node's functionality or to allow for new optimizations. Examples for such data are the location of neighboring nodes [Karp and Kung 2000; Minder et al. 2005] and information about their current role [Römer et al. 2004]. Typically, developers create application-specific protocols for this task. This approach tends to incur significant development overhead but offers the possibility of high efficiency with optimized protocols. Nevertheless, in real development projects, the effort for creating such optimizations might be too large and the actual solution might, thus, be suboptimal. Therefore, a general-purpose algorithm would not only reduce the development effort but – in some cases – also allow for greater efficiency. Although it is still possible to create more efficient application-specific solutions, our algorithm already provides good efficiency at no additional development costs.

In this section we describe our algorithm for neighborhood data sharing that strives to minimize the number of bytes transmitted while dealing with unreliable communication links. In addition, we use this algorithm as the basis of programming abstractions to facilitate the development of efficient applications that use data from neighboring nodes.

Although neighborhood data sharing only involves communication in a limited part of the sensor network and the size of such data is often small, the data of all nodes throughout the network adds up to considerable amounts. Therefore, optimizing such transmissions locally on each node can result in significant improvements regarding the number of messages sent and enhance the energy efficiency of the whole network. So far, however, most work has focused on disseminating data to all nodes in the network (e.g., [Heinzelman et al. 1999; Levis et al. 2004b; Hui and Culler 2004]) or on data-centric algorithms that transmit data to a sink node (e.g., [Intanagonwiwat et al. 2000]). In contrast, sharing data efficiently within the neighborhood has not been studied in sufficient detail yet. Even work dealing with programming abstractions for data sharing left the actual data transmission algorithm to be created by the application developer [Whitehouse et al. 2004] or only provided simple ones [Welsh and Mainland 2004].

There are two classes of data sharing algorithms: push-based and pull-based approaches [Franklin and Zdonik 1997]. With push-based approaches a node providing data sends it without having received an explicit request for it. Obviously, such approaches can lead to inefficiencies when the node's neighbors do not need this data. Especially in heterogeneous networks a node cannot necessarily infer what data its neighbors need because they may execute different code. Thus nodes might transmit unnecessary data or omit data that is actually required.

The second class of data sharing algorithms is composed of pull-based approaches. Here nodes only send data when they have received a request for it. This approach is better suited for heterogeneous networks, since each node may request the data it actually needs. The only shared assumptions are that neighbors can provide the requested data and use the same naming scheme. However, a pull-based approach can incur significant overhead for sending requests.

Therefore, we have developed *Neidas* ("**NEI**ghborhood **DA**ta **S**haring algorithm"), an efficient pull-based algorithm for neighborhood data sharing. Similar to network-wide dissemination approaches, our algorithm makes use of overhearing requests and data from neighboring nodes. It leverages the advantages of both pull-based and push-based strategies: The algorithm works well with heterogeneous networks and reduces the overhead for requests.

We have integrated *Neidas* into *TinyXXL* to create a comprehensive system for data sharing among components on a single node and on neighboring nodes. This system reduces the effort for the application developers because they do not have to create their own neighborhood data sharing algorithm and can use the same abstraction for both local and neighborhood data sharing. Furthermore, it combines the advantages of *TinyXXL* and *Neidas*: automatic optimizations, avoidance of redundant data storage and gathering, efficient and reliable data sharing.

```
(1) In each request round:
    Wait for the listen-only period and random interval
    For each data item needed from neighbors:
        If less than $k_r$ identical requests have been received:
            Send request
(2) In each data send round:
    For each data item requested by other nodes:
        If data item not requested in last data send rounds:
            Remove request
    Wait for listen-only period and random interval
    For each data item requested by other nodes:
        For local data and data received from neighbors:
            If less than $k_d$ copies of data with same version
            number have been received from this node:
                Send data including version number
    Double duration of data send round
(3) If new neighbors arrive:
    Reset duration of data send round to one request round
(4) Request received:
    Mark data as requested
    Increment counter for request
(5) Data received:
    If data requested and data is from node in neighborhood:
        If version number > stored version number
            Store data, source node, and version number
        Else if version number == stored version number
            Increment counter for this data
```

Figure 4.7.: Overview of the *Neidas* algorithm

4.4.1. Neighborhood Data Sharing Algorithm

Neidas is a data sharing algorithm that retrieves data from all neighboring nodes in radio range and continuously transmits updates when this data changes. It is based on the observation that – even in heterogeneous networks – there are typically several nodes within radio range that are interested in the same data. Therefore, our algorithm can take advantage of polite gossiping, which was first introduced in the Trickle algorithm [Levis et al. 2004b]. Trickle is an algorithm that efficiently distributes information about code images in the whole network. With its polite gossiping approach nodes wait a random time before sending data or a request for data from neighboring nodes. If during this time k_r neighbors send the same request, polite gossiping suppresses the transmission of redundant messages. Therefore, this algorithm leverages the broadcast nature of radio transmission: If several nodes have the same request, making each node send it would be unnecessary. Similarly, *Neidas* uses the same mechanism to locally forward the data provided by neighbors.

To deal with transmission failures and dynamic topologies *Neidas* periodically resends requests and data in so-called request and data send rounds. Fig. 4.7 gives an overview of the basic operation of the algorithm. The algorithm is called: periodically in each

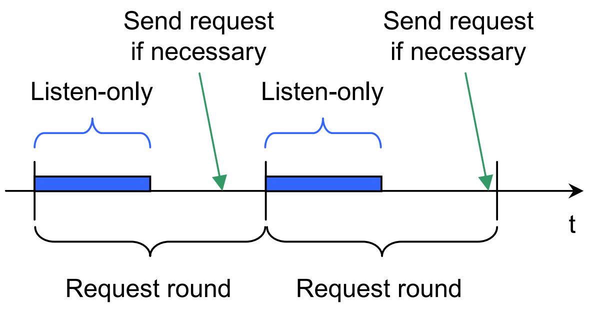


Figure 4.8.: Actions within request rounds

request round (1) and data send round (2), when new nodes arrive in the neighborhood (3), and when requests (4) or data packets (5) are received. The following subsections describe the *Neidas* algorithm in more detail.

Neighborhood Management

Since *Neidas* retrieves and stores data from neighboring nodes, it needs to know which nodes are in the neighborhood. Our current implementation includes an algorithm that intercepts all *Neidas* packets to build this neighborhood table. This algorithm does not incur any message overhead because it does not send any packets itself. If it has not received any packets within a predefined interval, it removes this particular node from the table.

If there is already an algorithm available that provides the required interfaces, it can be used instead to avoid duplicate data in memory. For instance, neighborhood information can be retrieved from the *TinyStateRepository* (see Section 4.3), SP's neighbor table [Polastre et al. 2005a], or accessed directly from the algorithm providing the data. To demonstrate this flexibility we implemented several such algorithms.

Sending Requests

Neidas is a pull-based algorithm, i.e., nodes send requests for data that they need. It takes advantage of overhearing messages by suppressing requests if other nodes have already sent the same one. The algorithm periodically resends requests to deal with dynamic neighborhoods and transmission failures. Therefore, it divides time into fixed-length request rounds, which are shown in Fig. 4.8.

Starts of rounds do not have to be synchronized on neighboring nodes. This can lead to an increased number of messages if nodes send their requests early at the beginning of a round. To avoid this problem, each round starts with a listen-only period [Levis et al. 2004b] in which a node just listens for messages from its neighbors (see Fig. 4.8). In the rest of the round each node randomly selects a point of time at which it will

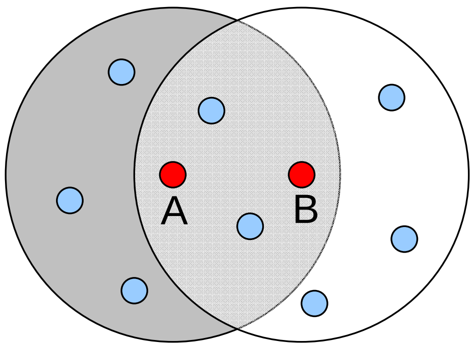


Figure 4.9.: Overlapping neighborhoods

send its request if by then it has not overheard at least k_r identical ones. Otherwise, it suppresses its own request in the current round.

Since neighborhoods may overlap, not necessarily all the neighbors of a node receive a request when the node overhears one. Therefore, a node might suppress its own transmission although not all of its neighbors have received the same request. For example, in Fig. 4.9 Node A sends a neighborhood data request which is received by all nodes in the dark circle. Therefore, Node B, which needs the same data, suppresses its own request. Thus not all nodes in its neighborhood (the light circle in the figure) receive the request. This is especially a problem because there might be no other node in the neighborhood requesting that data item. Trickle can easily deal with this issue since all nodes transmit the same kind of information and because version numbers ensure that the most recent information is always sent. *Neidas*, in contrast, addresses this problem in the following way. First, the threshold k_r is set to a slightly greater value than in Trickle. As simulations showed, a k_r value of 3 offers good results. Secondly, the random delay before sending a request ensures that not always the same nodes with the same set of neighbors send a request. Finally, following a soft-state approach for storing the requests with timeouts longer than a single round, nodes do not have to receive a request in every round. As shown in our simulations, in static topologies all nodes within radio range receive a request after some rounds. However, the algorithm still cannot guarantee that every single neighbor receives a message.

If communication links are asymmetrical, i.e., node A hears node B, but B cannot receive messages from A, *Neidas* remains functional, since it does not necessarily require direct interaction between nodes to request and transmit data – given that there are other nodes with the same request. Thus *Neidas* fully makes use of the broadcast nature of radio transmission with its polite gossiping scheme.

Sending Data

Besides sending requests, *Neidas* takes care of sending the requested data itself. Nodes transmit this data in two cases. First, they send all data matching a request periodi-

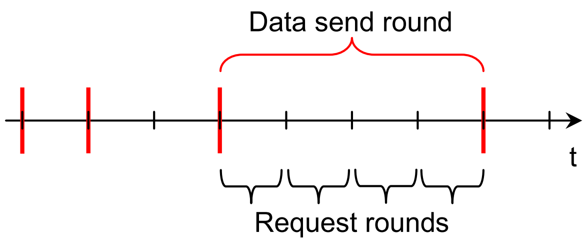


Figure 4.10.: Relation between request rounds and data send rounds

cally – including data received from neighboring nodes. This helps to make sure that after some retransmissions all neighbors have received it. As long as the request is valid, i.e., its soft state has not timed out, the data is resent. Second, when the local data is modified, nodes send additional updates to their neighbors. This way the neighbors receive the most current data even before the next regular retransmission.

For sending data *Neidas* also takes advantage of polite gossiping: Nodes that have received data from one of their neighbors transmit this data in addition to their local values. Since data is associated with a single node, only exactly the same data from the same node can suppress a transmission. In order to ensure that just the most current data is resent, the data includes a version number which is incremented whenever the data changes. To deal with version number overflows, receivers only accept data if this number is within a given range.

Since data is only relevant for the immediate neighbors and since only data originating from the same node can suppress its transmission, the polite gossiping threshold k_d for data can be smaller than k_r for requests. In addition, the version numbers define a prioritization where more recent data will not be silenced by older versions.

Nodes only accept data originating from one of the neighbors in radio range. This makes sure that data received via a third node is not disseminated throughout the network but kept within the neighborhood. Although *Neidas* currently only uses the radio range to define the neighborhood, with polite gossiping it would be easy to transmit data to differently defined groups of neighbors such as those proposed by abstract regions [Welsh and Mainland 2004].

Data is not sent in every round in which it has been requested. The reason for this is that *Neidas* tries to reduce the number of packets. Since the data itself is often somewhat larger than a request message, it is important to minimize the number of data transmissions. Therefore, we have introduced data send rounds. All data requested in the last and current data send round is sent if k_d neighboring nodes have not already transmitted the same data. As Fig. 4.10 shows, a data send round is composed of one or more request rounds. The length of the data send round is doubled after each round (up to a predefined maximum duration), in the figure from the length of one request round to four of them. It is reset to the length of a single request round when new nodes arrive in the neighborhood. This way these nodes receive prompt replies to their requests while greatly reducing the number of messages in static topologies. Note that changes to the length of the data send round are local to each node; they do not require any coordination among nodes. The length value reflects each node's estimate how often resends are necessary to make sure that all neighboring nodes receive a data item while keeping the rate of messages low.

We use a soft-state approach to remove requests after some time. Requested data, however, is not removed as long as the node stays within the neighborhood and there is enough memory available. So even after long data send rounds or several failed data transmissions, a node using *Neidas* still can access a previously received version of its neighbors' data from the local cache.

Further Optimizations

It is well known that radio communication consumes large amounts of energy [Shnayder et al. 2004]. In addition, there is also a significant MAC layer overhead associated with every packet. Therefore, reducing the number of messages is even more important than simply reducing the amount of data to be transmitted. In TinyOS and its standard MAC layer protocol [Polastre et al. 2004], for instance, the MAC layer preamble, the header and the checksum included in all packets add between 17 (full duty cycle of receivers) and 2,663 bytes (low power listening with 1 % duty cycle). Thus with a default data payload size of 29 bytes the overhead of sending a packet is between 58 % and more than 9,000 %.

Requests for neighborhood data and the data itself are expected to be comparatively small. Therefore, as an optimization *Neidas* accumulates several requests or data transmissions into a single packet. This is easily possible since *Neidas* uses small integer IDs instead of long names to identify the data and its type.

Sometimes even further optimizations are possible. Many applications and algorithms periodically send messages that do not fill the complete payload. Therefore, *Neidas* can take advantage of this free space by piggybacking its requests and data onto these messages. If the radio is operated in promiscuous mode, it does not even matter whether or not the packet is addressed to the same node as the piggybacked data, which – in our implementation – is always broadcast to all nodes in radio range. However, piggybacking is not feasible in all cases. For example, it is possible that the application does not send any data itself or that there are not enough free bytes available in the messages. Therefore, if after a time interval specified by *Neidas* the data has not been piggybacked, the piggybacking component sends a separate packet for this data.

This approach may incur some additional delays. During this time neighboring nodes might already have transmitted the same request, so that using polite gossiping it no longer has to be sent. Therefore, *Neidas* checks before actually sending the request if it is still necessary; otherwise, it cancels it.

```
 1 module DataAccessM {
 2   uses interface Timer;
 3   uses xldata RoleData as RoleDataLocal;
 4   uses xldata RoleData as RoleDataN[];
 5   ...
 6   event result_t Timer.fired() {
 7     uint8_t i;
 8     for (i=0; i<Neighbors.count; i++) {
 9       if (RoleDataLocal.role
10       == RoleDataN[Neighbors.nodes[i]].role) {
11   ...
```

Figure 4.11.: Accessing neighborhood data with *TinyXXL*

Our implementation works with all packets sent by any TinyOS-based applications and protocols because it replaces the TinyOS components which provide the so-called active message interface immediately above the MAC layer. For both the higher-level and the MAC layer component itself piggybacking is completely transparent.

4.4.2. Programming and Runtime Support

TinyXXL, as it is described in Section 4.3, has been developed for data exchange among the components of a single node. To create a comprehensive system both for this kind of intra-node data exchange and for neighborhood data sharing we extended *TinyXXL* to support accessing the data of neighbors and use *Neidas* in the *TinyState-Repository*.

If a component wants to access data of its neighbors, it has to declare this property as a dependency. Then it may use the neighbors' data similar to an array with the neighbors' node IDs. For instance, the code snippets in Fig. 4.11 show in line 4 how a dependency for role information [Römer et al. 2004] of neighboring nodes is declared. The brackets at the end of the line, which are not given for the dependency on the corresponding local values (line 3), denote that data is requested from neighbors and then accessed in an array-like fashion. With these declarations both local role information and that of neighboring nodes can be accessed (see lines 9 and 10 in the figure). If the data of a node is accessed which has not been received yet, a default value specified with the declaration of the data is returned (e.g., a reserved value indicating the absence of data). Since RAM is very limited on sensor nodes, *Neidas* does not store separately which nodes have already sent their data.

If data from neighboring nodes is declared to be accessed by at least one component, the *TinyXXL* compiler reserves some memory for caching this data locally. In addition, it adds calls to the *Neidas* algorithm to retrieve and continuously update the data. The compiler ensures that for each data item – even if it is requested by several components – there is only one such request sent and that the same data from a single node is stored only once in RAM. This way applications can benefit from the advantages of *Neidas* without adding the burden of implementing data exchange on the application developer. In fact, it is possible to retrieve some arbitrary data from the *TinyStateRepository*. The developer of the code running on the neighboring nodes does not have to be aware of the fact that an already existing piece of data might be needed by another node. This is an important advantage of our approach that facilitates independent development of software in heterogeneous sensor networks as well as reusability and exchangeability of components, whose data is automatically shared with neighboring nodes when necessary.

One inherent assumption of this solution is that on neighboring nodes the data is provided by a component and stored in the *TinyStateRepository*. Because of optimizations performed by the *TinyXXL* compiler, data is only gathered on a node if there is a component that needs to access it locally. Otherwise, it removes the data gathering code to reduce runtime overhead. In this case these nodes cannot answer requests for such neighborhood data. Therefore, like with manually implemented data sharing, the developer has to ensure that data needed from neighboring nodes is available.

If a node just accesses its local values and not those of its neighbors, there is almost no overhead associated with the integration of *Neidas*. In this case *Neidas* does not transmit any request messages. In addition, its RAM consumption is almost negligible: There is no need to reserve memory for local copies of neighborhood data, if the node does not use it. However, *Neidas* has to reserve two single bits for each kind of data in order to check if it has been requested by neighboring nodes within the most recent data send rounds.

Our solution offers the benefits of *TinyXXL* also for neighborhood data sharing. For example, it decouples components providing and accessing data: The component providing a piece of data that is needed by another node can be different from the one requesting data. In fact, in heterogeneous networks the component providing this data does not have to be part of the application requesting it. In addition, just like with local data it is possible to use virtual data items to transform data or perform some computations on it. Thus a neighboring node does not have to store a piece of data in order to provide it, as long as it can be converted to the target format. Furthermore, by taking advantage of the *TinyStateRepository*'s publish/subscribe mechanism the system can transmit updated data to its neighbors if it has been modified.

4.5. Evaluation

This section presents evaluation results for data exchange on a single node using *TinyXXL* as well as for neighborhood data sharing with *Neidas* and its integration in *TinyXXL*. Using real-world applications it shows the benefits for developers. In

Table 4.3.: Complexity of sample applications

Application	# components	# LOC
Sense-R-Us	116	18,714
TinyDB	176	29,559
AcousticLocalization	69	11,359

Table 4.4.: Number of changed lines of code (added, removed, and modified) and total numbers in the modified components

	Changed # LOC			LOC mod.
Application	Add.	Rem.	Mod.	components
TinyDB	391	332	297	6,318
AcousticLocalization	40	24	8	1,468

addition, this section presents the results of simulations that measure the run-time overhead on the sensor nodes including memory, processing, and message overhead.

4.5.1. Data Exchange on a Single Node

Development Costs

To show the development costs of *TinyXXL* we have created a nontrivial application and modified parts of two other ones available from the TinyOS CVS repository [TinyOS] to use it. These applications were already used for the analysis of cross-layer interaction in Section 4.2. They contain between 11,000 and 30,000 lines of code (see Table 4.3). Our newly created application is Sense-R-Us, which uses a sensor network to determine the position of research assistants and to detect the location and duration of meetings. The modified applications are TinyDB and AcousticLocalization. As already describe above, TinyDB provides generic query processing capabilities whereas AcousticLocalization determines the geographic positions of nodes using the difference in the speed of radio waves and sound. In TinyDB our changes to use *TinyXXL* focus on the components related to communication. In our view, those can profit most from a structured approach to cross-layer interactions. Nevertheless, the rest of the application also provides some opportunities to apply them.

Table 4.4 summarizes how many lines of code had to be added, removed, or modified to add data sharing and parametrization capabilities using *TinyXXL*. It also shows the total number of lines of code of all components that were modified. In TinyDB we have created 24 shared data variables and parameters, but in AcousticLocalization just 4 parameters, because in this application the components work mostly on internal data. Therefore, as Table 4.4 shows, in TinyDB by far more lines had to be changed. In general, the lines that were added are quite simple such as the variable declaration and the specification of data dependencies. Correspondingly, the lines of code that were removed were used to declare and share the variables with specialized

Table 4.5.: Lines of code in a minimal application and number of changed lines when adding another publisher

Version	Total # LOC	Changed # LOC		
		Add.	**Rem.**	**Mod.**
Pure nesC, data in publisher	46	1	9	1
Pure nesC, data separate	65	1	2	1
TinyXXL	39	1	0	1

interfaces. Modified lines of code were mostly changed so that accesses to shared data and parameters refer to the *TinyXXL* variables. One reason why the number of lines added is greater than those removed is that our modified versions publish more data than the original implementations because this data could also be useful for other components. In summary, however, there is significant effort needed when modifying existing applications to use *TinyXXL*.

If applications are developed from scratch with *TinyXXL*, the development costs are much smaller. In fact, there are fewer lines of code necessary than in pure nesC-based solutions. To show this, we have implemented a minimal application that shares a single variable between two components. We have created three versions of this application. Using just nesC, the first one stores the shared data in the publisher component (see Fig. 4.3(a) on page 62) whereas the second one stores it in a separate component (see Fig. 4.3(b)). Finally, the third version uses *TinyXXL* for data exchange (see Fig. 4.3(c)). All three variants offer the same functionality, use 20 bytes of RAM, and compile to 452 bytes of code.

Table 4.5 compares the code length of the different variants, which is one of the best predictors of understandability and maintainability [Sommerville 2001]. The results in Table 4.5 show that the *TinyXXL* variant requires the smallest number of lines of code. In particular, it needs 40 % fewer lines than the nesC approach that keeps the data in a separate component and still 15 % fewer lines than the nesC variant storing the variable within the publisher component. Although these numbers depend on the number of shared data variables, the number of accesses to them, as well as the number of publishers and subscribers, this example gives some idea about what we expect to find in more complex applications.

Sense-R-Us shows that complex applications can be developed with *TinyXXL*. This application makes extensive use of data sharing. For example, during the first 60 seconds after power-on – when information about neighboring nodes is gathered – data from the *TinyStateRepository* is accessed almost 3,300 times. In retrospect *TinyXXL* has made developing this application much easier.

Table 4.6.: Code size of the example applications (in bytes)

Application	Original	*TinyXXL*
TinyDB	62,144	61,894
AcousticLocalization	24,272	23,996

Maintainability

To show the benefits to maintainability, we modified the different versions of the minimal application introduced above. We added another component that provides the same data as the publisher already present. We then tried to optimize the application so that only one of the publishers writes the shared data and that no redundant variables are stored in RAM. As shown in Table 4.5, in all three versions of the application one line had to be added and one had to be modified. Besides that, in the two pure nesC versions code had to be removed manually because it would have been redundant. In the variant with the data stored directly in the publisher these were nine lines of code because the declaration of the redundant variable, all accesses, and the code providing the shared data to other components had to be removed. In the version that keeps the shared variable in a separate component only the accesses and the unused interface had to be deleted (two lines of code). In the *TinyXXL* version, however, nothing had to be changed.

Space Requirements

Table 4.6 shows the size of the code in program memory for both the original applications and the ones built with *TinyXXL*. The numbers are almost identical because most function calls to the *TinyStateRepository* are inlined by the compiler and, therefore, the code is mostly equivalent. Due to slight differences in the implementation and the optimizations performed by the compiler there are some variations (about 1 %). However, they are too small to derive a general trend.

TinyXXL does not increase the size of allocated memory in RAM. The data in the *TinyStateRepository* is the same as in pure nesC-based approaches, since the *TinyStateRepository* does not store any meta-data in limited RAM. If the application makes use of TinyCubus's adaptation capabilities, there is no RAM overhead for the *TinyStateRepository*, either, since all information about data providers and subscribers is written to the code in program memory. Only if support for neighborhood data sharing is included in the application, there is a slight memory overhead (see Section 4.5.2).

It should be noted that both TinyDB and AcousticLocalization have already been optimized by hand. Therefore, this setting is the worst case where *TinyXXL* is not able to add any benefit via its optimizations. With less optimized applications than our sample applications (e.g., those built from reusable components), we expect both code size and RAM consumption to decrease since the *TinyXXL* compiler includes

Table 4.7.: Runtime overhead of *TinyXXL*/ *TinyStateRepository*

	CPU Cycles		**Time (μs)**	
	Original	***TinyXXL***	**Original**	***TinyXXL***
Read access	13.36	16.06	1.81	2.18
Write access	19.81	19.27	2.69	2.61

only one instance of the data in memory and removes redundant data gathering code.

Runtime Overhead

To find out the runtime overhead of *TinyXXL* compared to a pure nesC approach, we measured the number of processor cycles needed for data exchange. For this purpose we used Avrora [Titzer et al. 2005], an emulator for Mica2-based sensor networks. We instrumented both the original and our modified versions of TinyDB and ran both versions of the application for 60 simulated seconds. During this time the nodes periodically exchanged messages and updated their routing tables several times.

The numbers in Table 4.7 show that the overhead of *TinyXXL* for read accesses is about 20 %. Although this overhead seems to be significant, it is not noticeable when running applications in practice; in absolute numbers it is just 0.37 μs. Furthermore, the speed of the processor and the clock cycles available are usually not regarded as a limiting factor in existing sensor network applications. In fact, compared to radio bandwidth the CPU is fast enough to process incoming messages without a need for receive queues [Levis et al. 2004a].

We attribute the overhead of read accesses mainly to fewer optimizations performed by the nesC compiler. Although most function calls can be inlined, there are some situations where in the original version of TinyDB a variable is already stored in one of the processor's registers and does not have to be loaded again. The compiler does not always perform this optimization with data from the *TinyStateRepository* so that the average cycle count is slightly increased.

For write accesses the numbers of both versions of TinyDB are almost identical. In fact, by coincidence the results for the *TinyXXL* version is even slightly better. In general, the compiler performs the same optimizations in all cases since it always has to write the value to the variable. Note that for this comparison the*TinyXXL*'s notification functions were not included since they provide some additional functionality. In any case, the overhead of these functions is not associated with every write access but with "**ifproviding**" blocks because the subscribers are only notified once for each block. In our modified version of TinyDB, for example, about 2.5 write statements are enclosed in such a block on average.

If support for dynamic adaptation within TinyCubus is needed, the optimizations usually performed by the *TinyXXL* compiler are done by the FlexCup linker. Since –

compared to the number of data accesses – applications are only adapted infrequently and since the linking process already takes a few seconds [Marrón et al. 2006a], applying these optimizations is not performance-critical. However, with this approach there is also some extra overhead at runtime: for each "**ifproviding**" block an additional check is required, because the code of such a block cannot be removed by the compiler in this case. In TinyDB this overhead is just 3.94 cycles (0.53 μs) on average. Because typically some computation or even the transmission of radio messages is included in such a block, the benefits of not executing this code – if it is not needed – clearly outweigh this overhead.

Even small processing overheads at runtime can sum up in the course of time and influence the lifetime of the sensor network. However, only if an application has been optimized, it provides better overall performance. Such manual optimizations are less and less feasible as applications become more complex and are increasingly developed by experts in the application domain rather than experts in sensor networks. Compared to unoptimized applications that gather the same kind of data twice, the small runtime overhead of *TinyXXL* and the *TinyStateRepository* can be neglected. For example, if *TinyXXL* avoids sending unnecessary radio packets, this alone will outweigh the energy consumed by the CPU for the *TinyStateRepository*'s small runtime overhead.

4.5.2. Neighborhood Data Sharing

Experimental Setup

We have simulated *Neidas* using the Avrora simulator. Unless otherwise noted, each simulation scenario contains 50 nodes which are randomly placed in a 60 m × 60 m rectangular area. Since communication is only local to the neighborhood, we expect that the results are also valid for larger-size networks.

The nodes' radio model is a lossy model, which is based on empirical data and has a transmission range of about 15 m. The TinyOS MAC layer takes care of multiple accesses to the radio channel. The measurements shown are the average of 10 runs of 600 simulated seconds each. We have set *Neidas*'s polite gossiping thresholds k_r to 3 and k_d to 1. As described above, experiments have shown that good results can be obtained with these values. The duration of a request round has been set to 10 s for all algorithms but in long-running experiments this value can be neglected – as long as all algorithms use the same duration. Nodes are turned on randomly in the first 10 s and are not switched off before the end of the simulation. Unless otherwise noted, we have not made use of piggybacking optimizations.

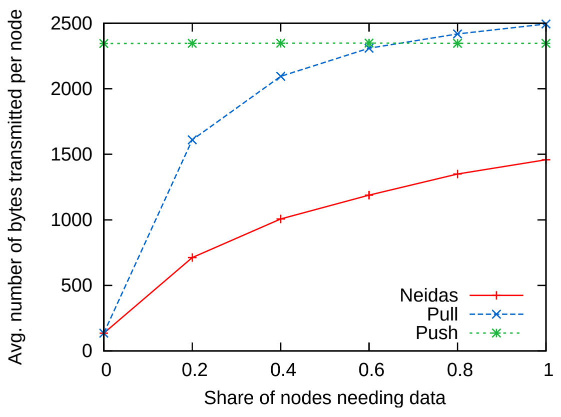


Figure 4.12.: Bytes transmitted, varying share of nodes requesting data

Efficiency of Neidas

We have created straight-forward implementations of standard pull-based and push-based algorithms, which are likely to be integrated in similar form in real applications. For a meaningful comparison, all of them use the same underlying data format and marshaling components as *Neidas*.

The pull-based algorithm periodically requests the data of neighboring nodes but does not suppress requests already heard. Similar to *Neidas* it does not service a request immediately but waits until the next round. However, it does not distinguish between request and data send rounds. The push-based algorithm periodically broadcasts its data without the need for requests. Neither of these algorithms resends data from neighboring nodes.

In our simulations all nodes provide a single data item of 10 bytes. We have varied the ratio of (randomly selected) nodes that needed this data. The only messages sent are those to request and transmit data.

Fig. 4.12 shows the total number of bytes transmitted by each node on average – including the packet header, preamble, etc. Since there are no big differences in the processing overhead of the three algorithms, overall energy consumption is dominated by the radio. Therefore, the energy consumed by the algorithms can be inferred from the number of bytes transmitted.

The push-based algorithm always transmits the same number of bytes because it does not distinguish between nodes that need data and those that do not. In contrast, for the pull-based algorithm the number of bytes transmitted grows with the percentage of nodes requesting data. If this percentage is greater than about 70 %, the pull-based algorithm is less efficient than the push-based one because of the additional overhead for request messages. Even when all nodes request data, the overhead for these requests is relatively small. The reason for this is that the pull-based algorithm uses the efficient underlying techniques from *Neidas* to build packets. Therefore, requests are usually

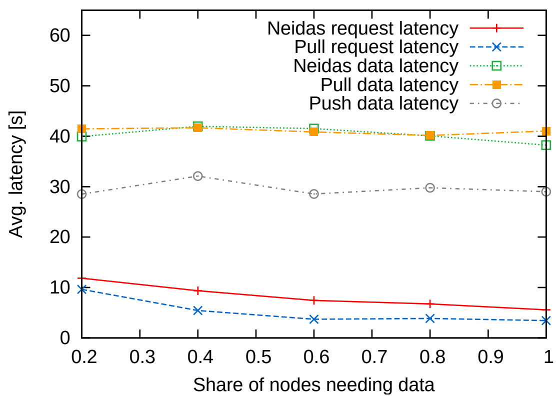


Figure 4.13.: Latency until requests and data have been received

sent together with the data as a single message, and there is no overhead for the packet header, preamble, etc. Otherwise, the numbers of the pull-based algorithm would be up to 500 % greater (not shown in the figure) because the payload is very small and thus the overhead of sending extra packets has even greater effects.

Neidas transmits much fewer bytes than these two algorithms. Depending on the number of nodes requesting data, it only sends between 30 % and 62 % of the number of bytes of the push-based algorithm and between 44 % and 58 % of the pull-based algorithm. Up to 20 % of the savings compared to the corresponding pull-based approach are due to polite gossiping of requests. This percentage increases with higher node densities. Enlarging the length of data send rounds is responsible for the rest of the savings.

Fig. 4.13 compares the average latency until a node entering the neighborhood receives requests and data. The request latency of *Neidas* is up to 4 s greater than that of the pull-based algorithm because of suppressed request messages in overlapping neighborhoods. However, when comparing the latency of the data itself, the values for both algorithms are almost identical because with *Neidas* nodes are able to provide also data requested from their neighbors. The data latency of the push-based algorithm, of course, is even shorter since with this algorithm nodes do not wait for requests before they send their data. The values for the data latency may seem comparatively high given the duration of the request rounds of 10 s. However, these numbers are average values until the data from *all* neighboring nodes has been received. Due to lossy links and collisions, some nodes have to send their data several times.

Fig. 4.14 shows the average number of bytes transmitted for different node densities. To get these values we have varied the total number of nodes from 25 to 200. The size of the area is kept constant and always 40 % of the nodes request data from their neighbors. The figure shows that the values for the pull-based algorithm increase by about 38 % with higher densities until all nodes are in the neighborhood of at least one node requesting data. For the push-based algorithm the number of bytes is constant

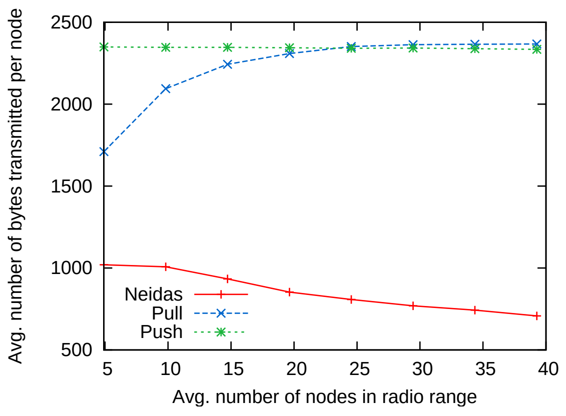


Figure 4.14.: Bytes transmitted varying the node density

since each node sends its data independent of other ones. With *Neidas*, the number of bytes transmitted by each node even decreases by about 30 % with higher densities although in these cases more nodes have to send their data. This is because more nodes overhear packets from their neighbors which avoids sending the same request several times.

As the results show, *Neidas* is suitable for both heterogeneous and homogeneous networks. Considering the benefits such as the small number of transmitted bytes shown in Fig. 4.12 and its ability to profit from high node densities (Fig. 4.14), for many applications *Neidas* offers a good compromise between efficiency and timely delivery of data.

Comparison with Hood

Hood [Whitehouse et al. 2004] is a programming abstraction that tries to ease neighborhood data exchange in sensor networks. By leaving the data transmission policies to be implemented by the developer, it allows for more flexibility than our system but increases the development effort. In this part of the evaluation we compare Hood with *Neidas* and *TinyXXL*.

We have implemented a simple algorithm that builds a tree to route data to a sink node with both Hood and *TinyXXL*. In this example all nodes request their neighbors' depth in the routing tree. They then select the neighbor with the smallest depth as their parent and adjust their own depth value. Using Hood we have implemented two versions: The first one minimizes the number of messages by solely relying on Hood's auto-push policy that only broadcasts updates when the data changes. The second one is able to deal better with new nodes and transmission failures of lossy links by periodically requesting the neighbors' values in addition to the automatic updates. This version resembles more closely the properties of *Neidas* but with its forwarding of neighbor data *Neidas* is able to provide even better reliability. Depending on

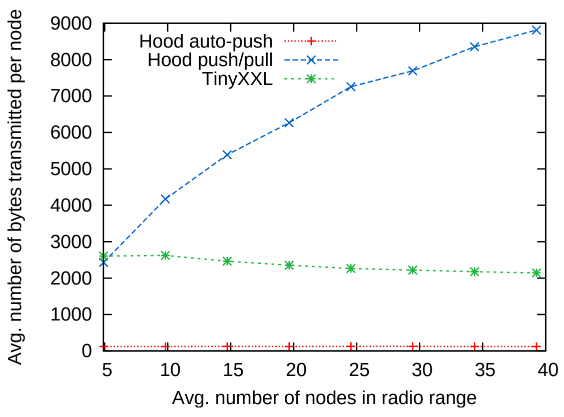


Figure 4.15.: Bytes transmitted compared with Hood

the properties required by the application, the solution actually implemented by the developer will probably lie somewhere within the boundaries defined by the two Hood versions. The *TinyXXL* implementation, however, automatically integrates *Neidas* so that the application developer does not have to deal with these low-level details.

Both the Hood versions and the *TinyXXL* version of the code use equivalent neighborhood management algorithms. These algorithms do not send any information themselves but use the node IDs transmitted with each request and data packet. It is an integral part of Hood's concepts that the neighborhood management algorithm has to be written by the application developer whereas in the *TinyXXL* version *Neidas*'s default neighborhood management algorithm is used.

Fig. 4.15 visualizes the total number of bytes sent by the Hood versions and the *TinyXXL* version for different node densities. Since the standard push-only version of Hood sends data just when it is modified, this algorithm transmits the smallest number of bytes. However, it is not able to deal with transmission failures and newly deployed nodes as it sends data only once. These two properties are fulfilled by the other two versions of the application. Therefore, these implementations offer different functionality and can hardly be compared. Thus we limit the following discussion to the *TinyXXL* variant and the Hood version including data pulls.

As expected, the number of bytes sent by *TinyXXL* decreases with high densities since it is based on *Neidas*. In contrast, the Hood version does not make use of overhearing messages and, therefore, has to transmit significantly more data if the number of nodes in radio range increases. For the highest node density the *TinyXXL* version sends only 24 % of the number of bytes transmitted by Hood.

When compiled for Mica2 nodes, our sample application including the operating system components reserves 810 bytes of RAM in the Hood versions and just 608 bytes in the *TinyXXL* version (25 % less). Most of *TinyXXL*'s savings are due to fewer variables used in the marshaling components as well as in the components storing data. With RAM sizes of just a few kilobytes such optimizations are crucial in order to be

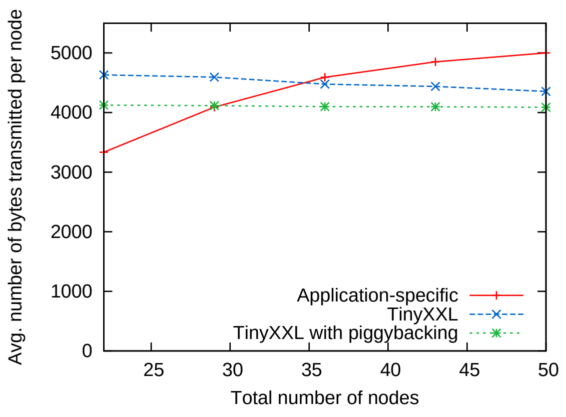


Figure 4.16.: Bytes transmitted for the Sense-R-Us application

able to create complex applications.

From a developer's point of view creating the application with *TinyXXL* incurs significantly less overhead. The routing tree algorithm described above was implemented in 176 lines of code with Hood (for the version including data pulls) vs. 88 lines of code with *TinyXXL*. This means that the *TinyXXL* implementation needs 50 % fewer lines of code than the Hood implementation. Most of these savings, however, are due to the fact that Hood requires the developer to implement a separate neighborhood management algorithm, which is already present in the *TinyXXL* solution. Although such numbers do not necessarily allow to draw conclusions about the complexity of the code, they can give a rough estimate about the effort needed by the application developer. Considering that Hood already reduces complexity compared to manual implementations [Whitehouse et al. 2004], the overhead reductions of *TinyXXL* are even more significant.

Integration in Sense-R-Us

In Sense-R-Us there are stationary nodes placed in rooms and mobile ones that are carried by persons. The mobile nodes use neighborhood data from the stationary ones to localize themselves by requesting information about the location of neighboring nodes. A mobile node's location is set to the value of a neighboring node, which has been selected using the received signal strength.

We compare an implementation of Sense-R-Us that has been built using *TinyXXL* with the original one for which neighborhood data sharing has been implemented manually. This version uses Sense-R-Us's custom querying protocol, which tries to reduce the number of messages by intelligently selecting the nodes to be queried. However, it does not leverage broadcast communication and comes at the expense of significant development overhead.

In our experiments we simulated up to 50 nodes of which 22 ones are stationary. The remaining nodes are mobile and move using a random walk model. Fig. 4.16 shows the number of bytes transmitted by the application-specific implementation of Sense-R-Us, a version using *TinyXXL*, and another *TinyXXL* version that takes advantage of the piggybacking optimization described in Section 4.4.1. These numbers also include packets transmitted by other components, e.g., to discover neighboring nodes. As the figure shows, for low densities with no or only few mobile nodes the performance of the *TinyXXL* versions is worse compared to the optimized application-specific solution. If, however, the node density is increased, the *TinyXXL* version can take advantage of overhearing messages and the number of bytes sent by each node decreases. For the application-specific implementation, in contrast, the number of bytes sent increases by almost 50 % when adding more mobile nodes. The reason for this is that this implementation relies solely on point-to-point communication. Therefore, separate messages might have to be sent even if other nodes have similar requests. If *TinyXXL*'s piggybacking optimization is used, the number of bytes transmitted is reduced by between 8 % and 13 % compared to the other *TinyXXL* implementation. These savings are due to the reduced number of packets sent by this variant. Although piggybacking could also be incorporated in an application-specific solution, using *TinyXXL* has the advantage that it comes “for free” without requiring the developer to manually implement it.

4.6. Related Work

This section describes work related to our classification of cross-layer interactions, to *TinyXXL* and the *TinyStateRepository*, as well as to *Neidas*.

There are several publications that give an overview of the use of cross-layer interactions. For example, Melodia *et al.* [Melodia et al. 2005] describe the use of cross-layer interactions in the sensor networks literature. Instead of identifying common mechanisms in these interactions, they rather group their examples by the interacting layers. A similar approach is pursued by Rasinghani and Iyer who give an overview of cross-layer interactions in 3G wireless networks [Raisinghani and Iyer 2004]. They describe protocols at different layers and group them based on their interactions with upper or lower layers, respectively.

Levis *et al.* [Levis et al. 2004a] describe some examples of cross-layer interactions in existing TinyOS applications. Although they give examples of different kinds of cross-layer interactions, they do not present a classification.

Finally, Srivastava and Motani describe cross-layer interactions in (general) wireless networks [Srivastava and Motani 2005]. They explain common mechanisms similar to our classification. For example, just as we describe in Section 4.2, they also identify merging of layers. However, there are also some differences. For instance, our classification focuses on the kind of information being exchanged (parameters or internal data) whereas theirs distinguishes cross-layer interactions based on the direction of data exchange in the protocol stack. In addition, Srivastava and Motani classify approaches for cross-layer frameworks. In this classification *TinyXXL* and the *TinyStateRepository* would be a shared database while the rest of our cross-layer framework does not fit in any of the categories provided.

Regarding work related to *TinyXXL* and the *TinyStateRepository*, blackboard architectures [Nii 1986] have been widely used in artificial intelligence systems. In this kind of architecture a blackboard is used to structure and store knowledge as a global database. Logically independent modules modify the data in order to incrementally solve a problem; they use the blackboard as the only form of interaction. Thus, similar to *TinyXXL* the modules are decoupled. However, there is no clear specification of data dependencies and all modules may modify the data. Furthermore, with *TinyXXL* there is still the possibility to employ the standard nesC forms of interaction.

In the realm of mobile ad-hoc networks (MANETs) the MobileMan project [Conti et al. 2004] creates a cross-layer architecture for the protocol stack. Although the concepts used for data sharing are similar to *TinyXXL*, MobileMan does not pursue the goal of easing the usage of cross-layer interactions and, therefore, is not part of a programming language. Furthermore, it assumes hardware platforms typical of MANETs, which are, in the general case, not so resource-constrained.

Part of the link abstraction SP for sensor networks [Polastre et al. 2005a] is a neighbor table data structure that can be accessed by protocols of several layers. However, with SP data sharing is limited to some *a priori* selected data items; unlike *TinyXXL* it does not allow for the definition of arbitrary shared data.

Köpke *et al.* [Köpke et al. 2004] have created a publish/subscribe-based system for sensor nodes. However, their work does not provide many of the features of *TinyXXL* because it is not integrated into a programming language. For example, it does not guarantee type safety when accessing data and does not deal with multiple publishers providing possibly inconsistent, duplicate data. Unlike our approach this system needs some meta-data to be stored in limited RAM.

Concurrently to our work Kumar *et al.* [Kumar et al. 2006] have developed a system with similar goals as ours. They decouple interacting layers by sharing data via the so-called Information Exchange Service, which is similar to our *TinyStateRepository*. However, their system does not seem to be as optimized to the resource constraints of sensor nodes as our system. For example, instead of using compile-time optimizations for memory efficiency their approach requires significant amounts of meta-data at runtime. In addition, the lack of static optimizations leads to increased access latencies, which this system tries to remedy with an memory-intensive caching mechanism.

TinyGUYS [Cheong et al. 2003] is a global data storage that deals with concurrency problems; it guards accesses to the variables by writing all changes in a buffer and having the scheduler copy this data to the real variables later. Such synchronization of accesses is not considered by our approach. We rather rely on the developer to add the standard nesC "**atomic**" statements when synchronization problems can occur. Obviously, TinyGUYS adds some memory overhead for the buffers which can be a problem with resource-constrained sensor nodes.

SNACK [Greenstein et al. 2004] is a configuration language, component library, and compiler based on the nesC language. Similar to our approach it tries to ease the development of efficient sensor network applications. However, it deals with the creation of service libraries that can be combined to form an application rather than focusing on the problems of cross-layer data exchange.

Our neighborhood data sharing algorithm is based on publish/subscribe-like interaction. Such systems have been used in different domains to make data available in the network. In sensor networks several algorithms following a publish/subscribe-like paradigm have already been proposed. Perhaps the best-known example is SPIN [Heinzelman et al. 1999], which uses such an approach to disseminate data in the network. However, these approaches typically require explicit interaction between every two nodes publishing and subscribing to data, which is not needed by our algorithm.

Gossiping algorithms are flooding-like approaches where nodes randomly forward data packets that they have received. Trickle [Levis et al. 2004b] uses polite gossiping to efficiently distribute information about code images in the whole network. It has been integrated into Deluge [Hui and Culler 2004], a code distribution algorithm, and adapted for the Drip protocol [Tolle and Culler 2005] to transmit queries to all nodes in the network. *Neidas* is inspired by the concepts of Trickle but deals with multiple nodes requesting potentially different data. Trickle, in contrast, can assume a single or few data sources and just one kind of data. In addition, with *Neidas* changes of data are kept local whereas Trickle disseminates them through the network.

As already described above, Hood [Whitehouse et al. 2004] is a programming abstraction that tries to ease neighborhood data exchange in sensor networks. However, it leaves important parts to be added by the application developer, e.g., data transmission policies that are responsible for sending data and requests. In addition, Hood does not strive to provide a comprehensive system for both intra-node and neighborhood data exchange.

Likewise, abstract regions [Welsh and Mainland 2004] provide programming primitives for local communication. An abstract region is defined using radio connectivity or the location of nodes, for example. Extending the neighborhood beyond immediate neighbors within radio range is something not considered by our approach yet. Like Hood, abstract regions only include a very basic data transmission algorithm. Similarly, logical neighborhoods [Mottola and Picco 2006] can be used for communication within a set of nodes that are not necessarily just the nodes in radio range. However, with this system a data sharing mechanism would still have to be implemented by the application developer based on other primitives.

There are numerous algorithms and applications that make use of data obtained from their neighbors. Most of them include custom solutions for neighborhood data sharing. Prominent examples are algorithms for self-organization [Römer et al. 2004], routing [Karp and Kung 2000], and medium access control [Ye et al. 2002]. By factoring out the transmission of data using *TinyXXL*, developers could focus more closely on the actual purposes of their algorithms and applications.

4.7. Summary

This chapter has presented several contributions. First, we have described our analysis of cross-layer interactions in typical sensor network applications and created a classification of the individual instances we found there. Second, based on this analysis, we have created *TinyXXL*, a programming abstraction for cross-layer data sharing and parametrization. *TinyXXL* strives to ease the development of cross-layer optimized applications built from reusable components. It allows for the declaration of components' dependencies on data, decouples components providing and using some data, and automatically optimizes applications by avoiding redundant data provision. Using *TinyXXL* the examples of Table 4.2 on page 60 for parametrization and data sharing can be implemented without coupling the interacting layers tightly. With *TinyXXL* data dependencies become first-class citizens similar to functional interfaces. *TinyXXL* is complemented by the *TinyStateRepository*, an efficient state repository generated by the *TinyXXL* compiler that stores all cross-layer data at runtime. Finally, we have presented *Neidas*, a pull-based algorithm for neighborhood data sharing, and its integration into *TinyXXL*. The combined system is a comprehensive system for both data exchange among components and neighborhood data sharing. Using *Neidas* as its basis it offers efficient data sharing at largely reduced development costs. For example, in heterogeneous networks the developer of a node providing data does not even have to be aware that this data might be required by another one.

As shown in the evaluation, complex applications can be developed using *TinyXXL*. For example, we have modified TinyDB and AcousticLocalization, two nontrivial existing applications, to make use of *TinyXXL* and developed Sense-R-Us from scratch with our programming abstractions. With our approach fewer lines of code are needed for data sharing and parametrization while profiting of additional benefits such as the selection of the "best" component to publish the data. We have also shown that the *TinyStateRepository* is an efficient implementation of a state repository regarding its runtime overhead. Furthermore, *Neidas* and *TinyXXL* allow for efficient transfer of data with neighboring nodes at largely reduced development costs.

In conclusion, *TinyXXL* provides an efficient system both for cross-layer data exchange among the components of a single node and for neighborhood data sharing.

5. Data Storage in Virtual Memory

This chapter presents the cross-layer framework's virtual memory abstraction. With this part of the framework one of the main reasons for cross-layer interactions – the severe memory constraints of sensor nodes – is relaxed. The chapter describes a heuristic that tries to optimize the memory layout using simulation traces. Using experiments with real-world applications, an evaluation shows that the heuristic is able to improve the performance of the virtual memory system. Finally, the chapter presents work related to this part of the framework.

5.1. Preliminaries

Traditionally, main memory has been a very scarce resource in sensor networks, and most sensor nodes are equipped with just a few kilobytes of RAM (for example, the Mica2 nodes from Crossbow only have 4 KB). However, several sensor network applications already require more memory than available on current sensor nodes. For instance, TinyDB [Madden et al. 2005] requires the user to select at compile-time which functionality should be included in the code image. This decision can later only be changed by installing a new code image on each sensor node. Likewise, the maximum size of an application in the Maté virtual machine [Levis et al. 2005] is strictly limited by the amount of main memory available. Therefore, in many applications cross-layer interactions are often used to partially alleviate the memory limitations.

As applications for sensor networks increase in complexity, RAM limitations will continue to cause difficulties for developers. For example, if applications perform more complex analysis than just the simple aggregation of most applications today, even nodes with more memory will not be able to satisfy future needs. As the experience from other domains shows, even with comparatively large random access memories there is always a shortage of main memory.

In traditional computing systems virtual memory has been widely used to address this problem [Silberschatz et al. 2002]. With virtual memory, parts of the contents of RAM are written to secondary storage when they are not needed. This mechanism is easy to use since the system takes care of managing the memory pages. However, current operating systems for sensor networks [Hill et al. 2000; Han et al. 2005; Dunkels et al. 2004; Abrach et al. 2003] do not support virtual memory. Only recently t-kernel [Gu and Stankovic 2006] has emerged as the first attempt to demonstrate that virtual memory can be useful in sensor networks as well.

Sensor nodes are equipped with flash memory as secondary storage, which – with a size of between 512 KB and 1 MB – is much larger than main memory. In many cases sensor network applications use flash memory to log sensor readings for later analysis. It is organized in pages of several hundred bytes that have to be written *en bloc*. Accessing it is much more expensive than accessing RAM: it takes several milliseconds to read or write a flash page whereas variables in RAM can be accessed in a few processor cycles. In addition, accesses to flash memory are comparatively expensive regarding energy consumption. Nevertheless, as we show in this chapter, this type of memory is appropriate for the implementation of virtual memory on sensor nodes.

The system model used by operating systems such as TinyOS is well suited to the use of virtual memory. First, variables and addresses are known at compile-time because all memory – except for the stack – is allocated statically. Secondly, since there is only one application running, the locality of reference [Denning 1970] is increased compared to multitasking systems and accesses are more predictable. Finally, since sensor nodes usually execute code directly from program memory, which is separate from RAM, in sensor networks virtual memory will only be used for data. Therefore, the additional overhead introduced by virtual memory does not arise with every single instruction but only with data accesses.

This chapter presents *ViMem* [Lachenmann et al. 2007a], a system that provides a virtual memory abstraction for TinyOS-based sensor networks and uses flash memory to extend the size of RAM available to the application. Since energy is a limited resource in sensor networks, *ViMem* tries to minimize the number of flash memory operations. It uses variable access traces obtained from simulations to rearrange variables in virtual memory at compile-time so that only a small number of accesses is necessary. As we show in the evaluation, this algorithm can help greatly to reduce the overhead of our virtual memory solution.

Using simulation data, the algorithm determines which variables are accessed frequently. It splits up complex data structures such as arrays or structs to examine each element individually. We define as such a data element an atomic part of a complex variable with a simple data type like “int”. The algorithm then groups such parts of variables that are often accessed together. Likewise, it tries to put variables that are accessed frequently on the same memory page so that it is likely to remain in RAM most of the time.

The memory layout is determined offline by a pre-compiler that modifies the code to redirect variable accesses to virtual memory. *ViMem* manages the flash pages in memory so that the developers do not have to deal with these low-level details and access variables simply as if they were stored in RAM. The only difference for the application developers is that they have to tag all variables that they want to place in virtual memory with a special attribute. This way they maintain full control of which variables are stored in virtual memory. For sensor networks this is important to allow the developers to keep variables permanently in RAM if they are used in time-critical functions.

Our solution has several benefits. First, it makes it possible to develop complex sensor network applications without having to restrict the functionality due to memory constraints. Secondly, although the developers control which variables are placed in virtual memory, accesses to those variables are transparent. Finally, through an intelligent placement of variables that takes into account the properties specific to sensor network applications, to hardware platforms, and to the development process *ViMem* is able to provide its functionality with minimal runtime overhead. We argue that such optimizations are essential to make virtual memory usable for the resource-constrained devices of sensor networks.

5.2. Design

This section first presents relevant characteristics of sensor networks, then lists our design goals, and finally gives an overview of the system design.

5.2.1. Sensor Network Characteristics

A number of characteristics of sensor networks have influenced the design of our virtual memory system. First of all, the hardware platforms used in typical sensor networks do not include any support for virtual memory. Therefore, the whole system – including address translation, the detection of page faults, and the page replacement policy – has to be implemented in software.

In addition, the behavior of flash memory vastly differs from other types of memory like magnetic disks and RAM [Gal and Toledo 2005]. For example, there is a large difference in the access speed for reading and writing. This difference also becomes apparent with the energy consumption of flash memory. Therefore, the number of write accesses has to be minimized. Table 5.1 shows the properties of the flash memory chip used in the Mica family of sensor nodes. For this hardware, writing a page to flash typically takes 4.5 times as long as transferring one to RAM. Even more important, we have measured that writing requires about 23 times the energy of reading. Furthermore, there is a limit on how often each flash page can be written. Thus some kind of wear leveling [Gal and Toledo 2005] has to be used in order to make sure that in the long run each page is written a similar number of times.

Since sensor networks consist of a large number of devices that are embedded in possibly inaccessible locations and offer only limited user feedback capabilities, simulation has become an important part of the sensor network development process [Titzer et al.

Table 5.1.: Properties of the Atmel AT45DB041B flash memory chip [Atmel Corporation 2005]

Property	**Value**
Page size	264 bytes
Number of pages	2048
Number of internal SRAM buffers for pages	2
Max. standby current	10 μA
Max. page read current	10 mA
Max. page write current	35 mA
Typical page read delay (measured)	3.6 ms
Typical page write delay (measured)	16.3 ms

2005]. Therefore, simulation can be used as a tool for optimizations. Such optimizations are important in order to meet the application's lifetime goals.

5.2.2. Design Goals

Our main objective for *ViMem* is *to provide a virtual memory abstraction on sensor nodes that minimizes energy consumption and the number of accesses to flash memory.* Taking into account the properties of sensor networks described above we have identified the following design goals in order to achieve this general objective:

- *ViMem* should not require hardware support for address translation, etc.
- *ViMem* should minimize the number of write accesses to flash memory for energy and efficiency reasons.
- It should be efficient for frequently used variables.
- The developer should be able to control which variables are placed in virtual memory.
- Accesses to variables in virtual memory should be transparent to the developer.
- *ViMem* should allow for the reuse of existing application and system components without requiring major modifications.

5.2.3. Design Overview

Like the other parts of our cross-layer framework, *ViMem* consists of two main parts: a compiler extension and a runtime component. The compiler extension redirects variable accesses to *ViMem*'s runtime system and determines the placement of variables on the memory pages. The runtime component takes care of loading and storing flash memory pages when they are needed.

Compiler Extension

Developers should be able to use variables in virtual memory just like those in RAM. However, since sensor network hardware does not directly support virtual memory, all access to data in virtual memory must be redirected to *ViMem*'s runtime component. Our system accomplishes this by using a pre-compiler that changes all such variable accesses. This pre-compiler modifies nesC source code. Although this approach is no real virtual memory system in the traditional sense, it offers similar benefits to applications and developers.

The developer maintains full control of which variables are kept in RAM and which ones are stored in virtual memory. Only those tagged with a special attribute are put into virtual memory. This way, variables that are used in interrupt handlers and other performance-critical functions can always be kept in RAM. In TinyOS's execution model these functions are called "asynchronous events". All other functions are executed in the task context which is less performance-critical, as interrupt event handlers can suspend such tasks [Hill et al. 2000; Gay et al. 2003].

The pre-compiler executes a memory layout algorithm to place variables on pages in virtual memory. Our software design allows for easy use of different such algorithms. The default algorithm, which is described in Section 5.3, uses access traces obtained from simulation tools to get information about the frequency of memory accesses. Doing all the processing offline minimizes the effort at runtime on the sensor nodes.

Runtime Component

ViMem's runtime system is responsible for the management of memory pages kept in RAM and for the provisioning of data to the application. The challenge here is to determine which memory page has to be replaced when another one is loaded from flash memory. Therefore, the algorithm has to predict which pages are most likely used again in the future. In addition, it has to consider the costs for replacing a page. If a page has not been modified, replacing this page is less expensive than selecting a page that has to be written back to flash memory.

This algorithm was not the main focus of our research. Therefore, for *ViMem*'s replacement policy we employ the Enhanced Second-Chance Algorithm [Silberschatz et al. 2002], which approximates a least-recently used (LRU) page replacement strategy. This algorithm stores two bits for each page in RAM: a recently used bit and a modified bit. It uses these bits to group each page into one of the four classes described below. Then it selects the first page in the lowest nonempty class to be replaced.

The lowest class consists of pages that have neither been modified nor recently used. These pages are the best ones to replace. If a page has been modified but not recently used, it is part of the second class. Likewise, the third class comprises all pages that are clean but have been recently used. Finally, a page that has been recently used and

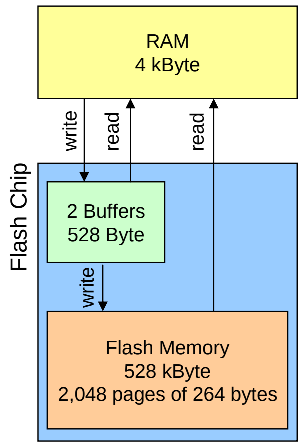


Figure 5.1.: Memory hierarchy of *ViMem*

modified belongs to the last class. It will not only have to be written back to flash memory but is also likely to be used again soon.

The page replacement algorithm uses a circular list to examine each page in memory. It starts looking for a page in the lowest category. If it finds none, it looks for one of the second class and then continues with the other ones.

In addition, the runtime system makes use of the SRAM buffers, which are part of the flash memory chip used in our implementation platform, the Mica2 nodes. Pages have to be transferred to one of these buffers before they can be written to the flash. In addition, as Fig. 5.1 shows, the buffers are used as a second level of caching: expensive write accesses are not performed immediately but the page is just stored in the buffer. If it is needed again, it does not have to be written to flash memory and can be retrieved from the buffer. Likewise, if it is not modified again while in RAM, it does not even have to be written back to the buffer again. If, however, a page is read that is currently not stored in one of the buffers, our system reads it directly from flash memory to RAM. This is possible without introducing additional overhead and leaves the SRAM buffers unchanged so that the number of actual write accesses to the flash can be further reduced. Since persistence is not required for a virtual memory system, it does not matter if a page in the buffer is lost when the batteries of the device are depleted, for example.

Each flash memory page can only be written a fixed number of times. Therefore, our runtime system tries to distribute write accesses across several physical pages in order to avoid wearing out pages that are written more often than others. For this purpose, it reserves a larger pool of flash memory pages than it actually needs (e.g., 1.5 times this number). Since there is usually enough free space available in flash memory and since we expect the space needed for virtual memory to be comparatively small, this does not severely reduce the flash memory usable by the application and other system components. If a page is written back to flash, *ViMem* cycles through a list of all reserved flash pages and selects the first one currently not being used. Following this approach, write accesses to frequently used pages will be spread across a larger number of physical pages. The data structures containing the information about available pages can be kept in RAM, as they are relatively small. In addition, since old contents of virtual memory do not have to be accessed after restarting a node, losing these data structures does not lead to problems. Of course, this aspect of *ViMem* is only needed if the flash memory chip does not take care of wear leveling itself.

5.2.4. Implementation

In this section we describe in more detail how *ViMem* has been implemented and how it can be used. We give this description for its two parts, the compiler extension and the runtime component, in the following subsections.

Like the rest of the cross-layer framework, our current implementation of *ViMem* is based on TinyOS and nesC. It has been optimized to the hardware properties of the flash memory chip used in the Mica family of sensor nodes. However, many of the concepts could also be applied in other operating environments and on other hardware platforms.

Compiler Extension

Overview of the Compilation Process Fig. 5.2 gives an overview of the compilation process of an application that uses *ViMem*. First, the nesC compiler checks the syntax of the source code and generates an XML file with information about components, variables, functions, etc. This file is then used by the *ViMem* pre-compiler to determine which components contain accesses to virtual memory and, therefore, need to be rewritten. The XML file also includes information about the data types of the variables that are to be placed in virtual memory. Exporting information about the program structure to an XML file has been introduced in version 1.2 of the nesC compiler [Gay et al. 2005]. Using information from the nesC compiler avoids duplicating existing functionality in our pre-compiler. Besides the source code, another input for the pre-compiler is a data access trace obtained from simulation. If this trace is available, it improves the performance of *ViMem* at runtime. However, it is not essential for building a running application.

Using all these inputs, the *ViMem* pre-compiler modifies the source files that access virtual memory and creates an optimized memory layout as well as components that make this layout available at runtime. These results, the *ViMem* runtime components, and other unmodified source files are the input for the actual compilation step, which is performed by the nesC compiler. As usual, this step results in a code image that can be installed on sensor nodes or simulated using appropriate tools. These two

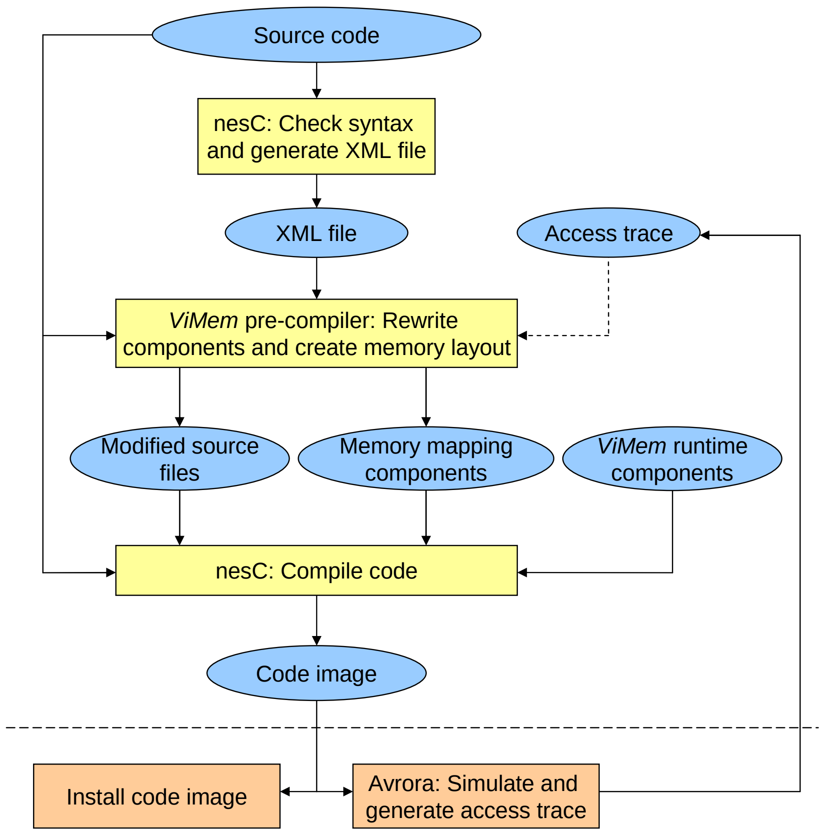

Source code
nesC: Check syntax and generate XML file
XML file
Access trace
ViMem pre-compiler: Rewrite components and create memory layout
Modified source files
Memory mapping components
ViMem runtime components
nesC: Compile code
Code image
Install code image
Avrora: Simulate and generate access trace

Figure 5.2.: *ViMem* compilation process

alternatives are shown in Fig. 5.2 below the dashed line because they are not part of the compilation process itself but are invoked separately.

If the developer chooses to simulate the application, the output of the simulator, in our case the Mica2 simulator Avrora [Titzer et al. 2005], can be used to generate an access trace. Avrora provides fine-grained instrumentation capabilities which allow to get detailed information about the execution without changing the timing of the program [Titzer and Palsberg 2005]. Using this instrumentation interface, we have created probes that record all accesses to data elements in virtual memory. The result given by these probes forms the access trace for the *ViMem* pre-compiler. If the application is compiled again, a memory layout optimized for the given access trace will be generated.

Implementation Details The pre-compiler has been implemented in Java using JavaCC as a parser generator. It modifies all components that access variables in virtual memory, creates the memory layout, and generates components that map variables to their actual position in virtual memory.

As the code example in Fig. 5.3 shows, all variables that are to be placed in virtual

```
uint16_t varInVM @vm();
uint16_t* pointer @vmptr();
uint16_t varInRAM;

uint16_t* testFunction(
    uint16_t* value @vmptr()) @vmptr() {
  *value = 54;
  return value;
}

command result_t StdControl.init() {
  varInRAM = 123;
  varInVM = varInRAM;
  pointer = &varInVM;
  varInVM = *testFunction(pointer);
  return SUCCESS;
}
```

Figure 5.3.: nesC code that accesses variables stored in virtual memory

memory have to be tagged with the attribute “@vm”. The ability to tag variables, parameters, and functions with user-defined attributes is another feature introduced in nesC 1.2. Only variables which are declared globally within a component or marked as “static” can be placed in virtual memory. Local variables of functions, in contrast, are always allocated in RAM on the stack.

It is not possible to reference a variable in virtual memory using a normal pointer variable, since its actual position in RAM may change after it has been swapped out to flash memory. In this case the runtime system could not associate the pointer value with a variable in virtual memory. Therefore, pointer variables as well as parameters and return values of functions that refer to a variable in virtual memory have to be tagged with the attribute “@vmptr”. All variables and parameters that are tagged with this attribute are modified by the pre-compiler as well: They no longer refer to a location in RAM but contain an ID that denotes the corresponding element in virtual memory.

The only restriction for the developer is that casting variables to types of a different size is not allowed. The reason for this is that data elements in virtual memory are not necessarily contiguous. Without this restriction it would be possible that several flash operations are necessary for such an access.

Adding the attributes “@vm” and “@vmptr” is the only change to the code that has to be performed by the application developer when using *ViMem*. As the example in Fig. 5.3 shows, accesses to such variables look exactly like accesses to normal variables. In the example there is one variable stored in RAM (“varInRAM”), one stored in virtual memory (“varInVM”), and a pointer to a variable in virtual memory. The sample code shows that *ViMem* uses pure nesC code. If the pre-compiler is not run,

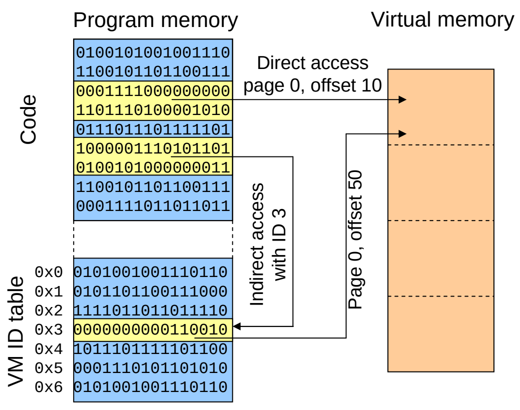


Figure 5.4.: Mapping of IDs to virtual memory

nesC could still compile the same code and store variables in RAM.

The *ViMem* pre-compiler takes this code and removes the declaration of all variables placed in virtual memory. Instead it assigns them an ID that refers to a position in virtual memory. Therefore, it replaces all references to such variables as well as accesses via “@vmptr” variables with calls to the *ViMem* runtime component that refer to the element’s ID. Like most functions in nesC, calls to the components of the *ViMem* runtime system are usually inlined. This reduces the overhead associated with accesses to variables in virtual memory. Furthermore, as described in Section 5.3, *ViMem*’s pre-compiler places variables intelligently in virtual memory in order to reduce accesses to flash memory.

For efficiency reasons, the pre-compiler uses two different ways to translate an ID of a data element in virtual memory to the right memory location. If the ID is known at compile-time, the pre-compiler inserts a direct call to the runtime system with the correct page and offset in that page. This is possible for all variables except pointers and accesses to arrays when the element index is stored in another variable. For these two types of accesses the exact ID of the element is not known at compile-time. Therefore, the runtime system has to look up the page and offset of the data element whenever such a variable is accessed. This information is stored in a table in program memory where it can be read efficiently without occupying space in RAM.

These two alternatives are shown in Fig. 5.4. The first ID can be resolved at compile-time and, therefore, the address in virtual memory can be directly inserted. For the second access, however, the ID needs to be resolved at runtime. For example, the data element could be accessed via a pointer. Thus the actual position in virtual memory has to be looked up in the table containing the IDs.

If possible, the pre-compiler uses the first type of access because the performance of the second variant is slightly worse (see Section 5.4.1).

Runtime Component

The runtime component is responsible for checking if a page needed currently resides in RAM and to move pages from and to flash memory. It has been optimized regarding its fast path, i.e., the overhead for the most common case (accesses to pages already in RAM) has been reduced. For accesses to pages that have to be loaded first, optimization is not as critical since a much longer delay is imposed by the flash memory operations. Furthermore, we have tried to keep the RAM consumption of the runtime system low. However, where possible, we opted to reduce overhead on the fast path by keeping data structures for efficient accesses in RAM (see Section 5.4.3 for details on RAM overhead). Since with *ViMem* the strict RAM limitations are no longer a severe problem, marginally increasing the RAM consumption of the system does not limit the memory available to the application.

The performance of *ViMem* depends on the number of virtual memory pages that are kept concurrently in RAM. This parameter is determined by the pre-compiler dynamically based on the memory space available. In Section 5.4.3 we show how this number influences the overall performance.

For accesses to flash memory, *ViMem* uses the standard TinyOS "PageEEPROM" component which has been slightly modified and extended for our purposes. Nevertheless, our version of this component is fully compatible with the old version, which also allows other parts of an application to access the flash. The only drawback of such uses by another component will be a somewhat degraded performance since the flash chip is blocked if that component reads or writes pages. In addition, *ViMem* has to share the flash buffers with that component, which may lead to an increased number of actual writes to the flash chip.

Our modifications to the flash component are limited to three changes: First, the flash component resets its state immediately after executing a command instead of doing this in a separate nesC task. This modification allows subsequent accesses to flash memory from a single task. Secondly, we optimized read accesses by transferring pages directly to the CPU without loading them into one of the flash chip's SRAM buffers. Since modified pages would have to be written to actual flash memory if another page was loaded into its buffer, this change reduces the number of write accesses to flash memory. Finally, we added a new command that avoids unnecessary page transfers from RAM if the same (unchanged) page is still stored in one of the buffers. This modification reduces the number of page transfers to the flash memory chip if modified pages have been copied from a buffer to RAM again only for reading.

5.2.5. Integration with TinyXXL and TinyCubus

This section describes how *TinyXXL* makes use of *ViMem* to store its data and which changes to *ViMem* are necessary if it is used within the dynamically adapting Tiny-

Cubus framework.

As described in Chapter 4, cross-layer data declared with *TinyXXL* is stored in the *TinyStateRepository*. If data from neighboring nodes is needed, it can amount to significant sizes. Therefore, *ViMem* lends itself to store the data of the *TinyStateRepository*. If the application developer decides to use it for this purpose, however, some data might be accessed from functions called by time-critical interrupt handlers. If the corresponding memory page has to be read from flash memory first, the application would probably not work as expected. Information about the functions called from interrupt handlers is already gathered by the nesC compiler. However, it is not exported to other tools. Therefore, we extended the nesC compiler to make this data also available in its XML file.

This is the only modification to the nesC compiler for our cross-layer framework. If it is not applied, it is still possible to use *TinyXXL* and *ViMem*. In that case, however, the developer should manually avoid accessing data via *TinyXXL* in time-critical functions.

TinyXXL's virtual data items (see Section 4.3) can still be used in the presence of virtual memory. Since their values are computed at each access, they are not stored in virtual memory themselves. However, the data from which they are derived can be placed in virtual memory.

If *ViMem* is used within the TinyCubus framework, some changes to *ViMem* – as it has been described above – are necessary.

If TinyCubus dynamically adapts the components forming the application, the memory layout has to be adjusted as well. Otherwise, it would not be optimized to the actual application running. In fact, in most cases the data elements required by the new component will not match those present in the old one's memory layout. Including a separate memory layout with each component is not feasible because then the overall layout of the application could not be optimized. Thus a combined memory layout for each set of components forming an application has to be created offline. Creating such an optimized layout on the sensor nodes is not practicable because of the additional runtime overhead. Therefore, there has to be a way of transmitting a memory layout suitable for the current components to the sensor nodes. Existing algorithms for code distribution cannot be used because they usually disseminate the code throughout the network. In the TinyCubus scenario, however, each node – or each group of nodes – can potentially run a different set of components. In addition, once the memory layout is available on the sensor node, it has to be integrated with the code image. Therefore, this task has to be done on the sensor nodes as well.

Besides the memory layout, the code of the components needed for adaptation has to be somehow transmitted to the sensor node. However, transmitting these components to all nodes of the sensor network would impose a large overhead. Therefore, we have developed a protocol that sensor nodes can use to request both the code and the memory layouts [Weinschrott 2007]. Since radio transmission is energy-expensive and since the size of such data can be several kilobytes, the protocol caches the data items in the network to reduce traffic. Nodes that are on the route and, therefore, receive the data anyway, decide themselves if they want to add it to their cache. If a node needs a new component or memory layout, it starts an expanding ring search [Yavatkar et al. 1995] to find a cache that is nearby.

To install the memory layout on the sensor node FlexCup [Marrón et al. 2006a], our dynamic linker, can be extended. Since FlexCup modifies the code before installing it into program memory, it is the natural place to update the memory layout. However, updating the layout in the compiled files requires some modifications to the code generated by the *ViMem* pre-compiler.

Having FlexCup modify the code itself, i.e., changing the parameters of the functions calling *ViMem*'s runtime system, is most of the time not feasible. These functions would have to be treated separately from any other functions processed by FlexCup and it would have to be ensured that the compiler does not do any optimizations that use the values of the parameters for any other purpose. Therefore, when using *ViMem* together with FlexCup not just arrays and pointers (as described in Section 5.2.4) but all variables have to be accessed indirectly using a reference to the memory layout in program memory. Although this is slightly less efficient, the additional delay is only noticeable for the access time if the data element is currently available in RAM. Therefore, FlexCup does not have to modify every access to virtual memory but simply replaces the table containing the memory layout mappings.

When combining components dynamically, the compiler does not have a global view of the variables available at runtime. Therefore, not only is finding an optimal memory layout impossible, but also assigning IDs to data elements is difficult. Since the set of components used at runtime is not known in advance, the ID for each data element would have to be globally unique, i.e., for all possible combinations of components. Even if assigning such IDs is possible (e.g., using a combination of a component ID and its local data IDs), they will be considerably longer than the two bytes normally used by *ViMem*.

It is necessary that within each application the IDs are unique. If they were just unique within each component, pointers to virtual memory could not be exchanged among the components. Therefore, we use the following solution for the integration of *ViMem* and FlexCup: Each component contains a second table in program memory that maps the locally unique IDs of data elements to the IDs used by the overall application. This table is also updated by FlexCup when installing the application. This solution requires an additional indirection. However, since the table is stored in program memory, the costs for that are – again – comparatively small.

Fig. 5.5 summarizes how all accesses to virtual memory are done when components can be dynamically replaced. Each component includes a separate table that maps its local IDs to the memory IDs of the application. The retrieved application ID is then used to look up the position in virtual memory. Both of these mappings are updated

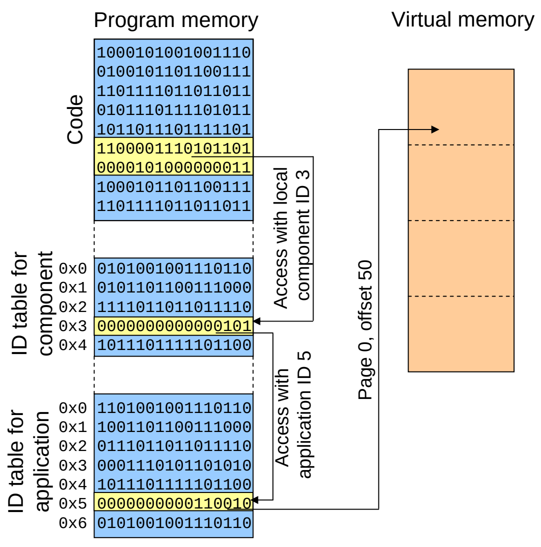


Figure 5.5.: Additional indirection for VM accesses within TinyCubus

when a new combination of components is installed.

In summary, FlexCup has to fulfill two additional tasks with *ViMem*. First, it has to update the table of each component that maps the local IDs of data elements to those used by the application. Second, it has to update these application IDs to reflect an optimized layout.

5.3. Memory Layout Heuristic

This section describes our memory layout heuristic that determines the placement of variables in virtual memory. It is the core part of our approach to reduce the number of flash accesses and, thus, improve on efficiency. The heuristic has two main goals: First, it aims to exploit locality of reference in order to reduce the overall number of page replacements. Second, it puts special effort in decreasing the number of write accesses to flash memory.

ViMem's pre-compiler runs this algorithm when building the application. Fig. 5.6 gives an overview of the heuristic. As we describe in the following subsections, it uses simulation traces to determine an efficient memory layout by grouping variables and placing them on memory pages.

5.3.1. Use of Variable Access Traces

In general, finding an optimal memory layout is not possible since the exact order in which variables are accessed at runtime depends on many factors. For example, in

```
Process variable access trace
While trace not completely processed
  Let $d$ be the currently accessed data element and $v$ the corresponding node in the graph
  Add $d$ to list $l$ of recently accessed elements
  If $l$ already includes another access to $d$
    Remove older entry
  While the size of all elements in $l >$ Constant
    Remove oldest entry
  Increment the weight of $v$
  Increment the weight of all edges between $v$ and the other elements in $l$

Group data elements
While not given percentage of elements grouped and suitable edges are available
  Let $e_{max}$ be the edge that connects $v_1$ and $v_2$, where $v_1$ and $v_2$ consist of less than a predefined
    number of data elements and where the normalized proximity $g(e_{max})/(f(v_1) + f(v_2))$ has the
    greatest value
  Merge $v_1$ and $v_2$ into $v_{group}$
  Set node weight of $v_{group}$ to the average of the elements it contains
  Remove duplicate edges from $v_{group}$ to any other node and set the edge weight to the maximum
    weight of these edges

Place groups on memory pages
While not all data elements have been placed
  Let $g$ be the group of data elements with the maximum number of accesses that has been
    assigned to a flash page yet
  If $g$ is written often
    Select first write-often page with enough space for $g$
    Place $g$ on that page
  Else
    Select first mostly-read page with enough space for $g$
    Place $g$ on that page
```

Figure 5.6.: Overview of the memory layout heuristic

sensor networks data gathering requests from users as well as sensory input and packets received from other nodes may influence the application flow. Even if the exact order of accesses were known at compile-time, finding an optimal memory layout would be an NP-complete problem [Gupta 1991]. Therefore, our memory layout algorithm can only provide a heuristic that does not necessarily find the best solution.

Although the specific order of data accesses is not predictable, usually there are patterns that recur. For example, some variables are often accessed following each other and some of them are accessed more frequently than others. Our heuristic uses simulation traces to determine such patterns for variables stored in virtual memory. Even if the same sequence of accesses is not repeated when running the application later, we argue that these traces can provide valuable hints for data placement. However, the simulation scenario has to resemble the actual operation of the sensor network. Since simulation is a technique often used when developing sensor network applications and since simulation scenarios have to be realistic to evaluate the functionality and performance of the application anyway, this step does not impose excessive additional burden on the application developer. Furthermore, as these results can be obtained by extending the simulator rather than modifying the application, gathering information about variable accesses does not alter the behavior of the application itself. Therefore, a single simulation run can be used both to test the application and to obtain a data access trace.

The *ViMem* pre-compiler can be configured to use for variable placement either the access traces of all nodes of the network, those of a group of nodes, or just the ones of a single node. This allows the system to optimize the memory layout for nodes with different tasks, although they may execute the same code image. For example, a node at the edge of the network can have different variable access patterns than a node at the center where more packets have to be forwarded. We expect that the performance of *ViMem* is best if a separate code image with an optimized memory layout is installed for each such group of nodes. Otherwise, the system remains fully functional, but at increased memory access costs.

If no simulation data is available (e.g., when building a new application), the *ViMem* pre-compiler uses the variable references in the source code to estimate the number of accesses. Obviously, this information can be inaccurate because it is unclear how often a function is called or which branch of an if-statement is selected at runtime, for example.

We have verified each design decision taken for our heuristic using existing sensor network applications and always selected the alternative that offered the best performance. As Section 5.4 shows, the results of the heuristic are promising.

As described above, the pre-compiler splits up complex variables, such as large arrays and structs, and examines each part individually. For example, the first elements of an array might be accessed more frequently than the last ones. Therefore, instead of recording the access just for complex variables as a whole, all data accesses are associated with individual data elements. This approach allows the memory layout to more closely resemble the actual access patterns. The only exception to this rule are small arrays: our algorithm always treats accesses to one of their elements as accesses to the complete array. In our experience with existing applications the elements of such arrays have a tight coupling and, therefore, should be regarded as a single entity.

5.3.2. Grouping of Data Elements

Having gathered information about accesses to data elements, the memory layout algorithm groups those data items often accessed together. When reading an access trace the pre-compiler calculates the weights of a fully-connected graph $G = (V, E, f, g)$, where the nodes V are the data elements and the edges E represent the relationship between the data elements. In this graph both the nodes and edges are weighted: The weight of a node, given by $f : V \rightarrow I\!R$, indicates how often the corresponding data

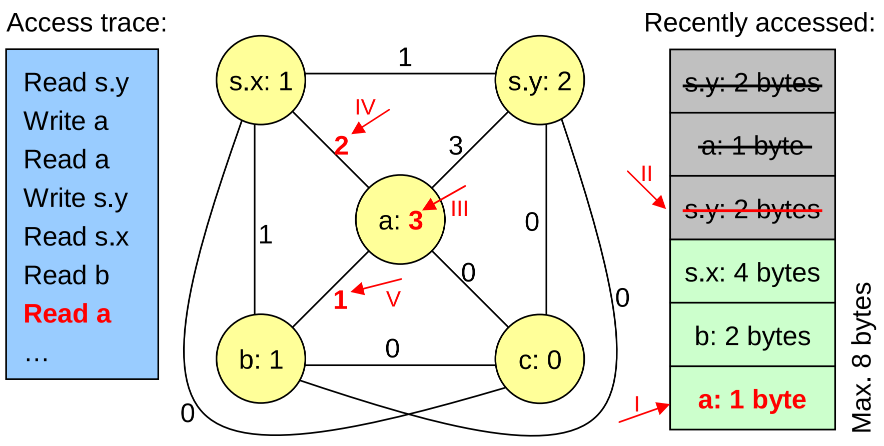


Figure 5.7.: Example for processing an access trace

element has been accessed, and the weight of an edge, defined by $g : E \rightarrow I\!R$, gives information about the proximity of the data elements it connects.

For each sensor node in the network, the pre-compiler maintains an ordered list of data elements that have been accessed recently. Each data element appears in this list at most once for only its most recent access. The sum of the sizes of all elements may not exceed a parameterizable constant. These elements represent those that should be preferably in RAM when the new element is accessed. If one of them is not stored in RAM, a page fault can occur and another page would have to be loaded to RAM. When the *ViMem* pre-compiler adds a data access from the trace, it increases both the access count in the data element's node and the proximity to all data elements in the list.

Proximity of two data elements deliberately does not take into account the temporal distance between accesses to data elements: To determine if they should be placed on the same memory page, it does not matter whether or not there is some delay between accesses – as long as no other variables are accessed in between.

Fig. 5.7 shows an example how an access trace is processed. The figure displays parts of an access trace for one node, the graph G, and the list of recently accessed elements. For simplicity, it assumes that the size of a memory page is just 8 bytes and that the same maximum size is used for the elements in the list. The figure shows the simple variables "a" (size: 1 byte), "b" (2 bytes), and "c" (4 bytes) as well as struct "s" with its fields "x" (4 bytes) and "y" (2 bytes). As described above, the algorithm splits up the parts of the struct and examines each field individually. In the example the last line of the access trace is being processed, which leads to the changes highlighted with arrows. First, the element is added to the list of recently accessed data elements (I). Since the total size of the elements in this list is greater than the page size, the algorithm removes the oldest element ("s.y", crossed-out in the figure, II). Then it increments the weights of "a" (III) and of all its edges to elements in the list (IV and V).

After reading the complete access trace, *ViMem*'s pre-compiler tries to group the elements that are often used together. To achieve this goal it merges nodes in the graph G that have high proximity values. This step is inspired by the procedure sorting algorithm [Muchnick 1997] that performs similar operations on the call graph to place the code of a procedure always near its callers.

In this step the algorithm does not use the raw proximity value $g(e)$ but normalizes them using the access counts of the data elements ($f(v_1)$ and $f(v_2)$) it connects: $p = g(e)/(f(v_1) + f(v_2))$. This way, the algorithm can form groups both of elements accessed frequently and of those only used seldom. It repeatedly selects the edge e_{max} from the graph G with the highest normalized proximity value p_{max}. The algorithm then merges the nodes connected by e_{max} and coalesces their edges. It sets the node weight of the group to the average of its elements' weights, which helps to treat merged nodes and original ones as equal when computing p. The weight of coalesced edges is set to the maximum of the original edge weights. This preserves close proximity between elements even if they have become parts of merged nodes. The elements forming one such new node are grouped and always placed on a single memory page later.

The algorithm repeats this process until a given percentage of data elements has been grouped. If, however, the size of the group exceeds a given limit (e.g., one eighth of the size of a memory page), no more elements are added to it. The reason for this is that we want to avoid very large groups which are less flexible when creating the memory layout.

5.3.3. Data Placement

After determining the groups of data elements used together, the memory layout algorithm places them on actual memory pages. This part of the heuristic processes the elements in the order of their access frequencies because proximity has already been exploited when forming groups. This way data elements that are accessed often are placed on the same memory page, which can probably stay in RAM for most of the time, while elements that are used only seldom do not occupy space on pages in RAM.

The algorithm places the data using a first-fit strategy with two sets of pages: one with elements that are modified often and one with those that are written only seldom. When placing an element, it checks if the number of write accesses is above a threshold. In this case the element is placed on a page that contains other elements that are written frequently whereas all elements that are mostly read are placed on different pages. If a page has to be removed from RAM, this approach makes it more likely that the page does not have to be written back to flash memory. Similarly, if a modified page is removed from RAM, it is probably written because of multiple changes. Therefore, this scheme takes into account the differences between read and

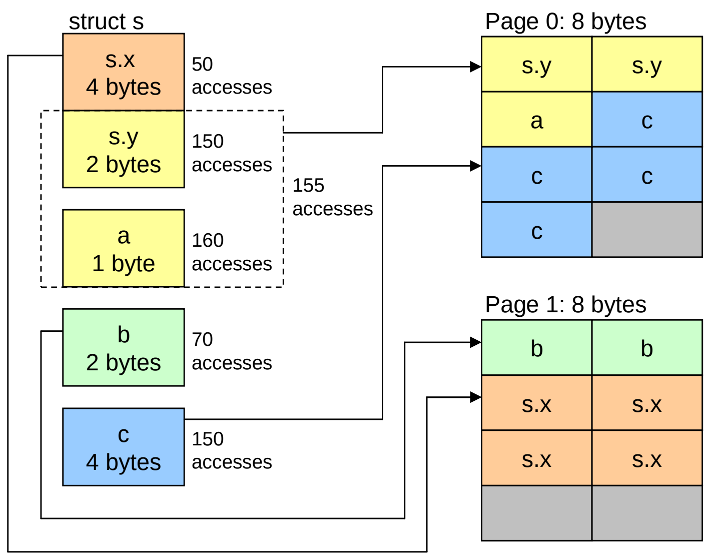


Figure 5.8.: Example for the memory layout algorithm

write accesses mentioned above. In some cases this part of the heuristic helped to reduce the number of write accesses by up to 70 %.

Fig. 5.8 takes up the example from Fig. 5.7 after the complete access trace has been processed. For simplicity, the example does not distinguish between read and write accesses. In the situation shown just "s.y" and "a" are grouped. Then all groups and single data elements are placed on flash pages in the order of their access counts. The variable "b" is put completely on the second page, as elements may not span several pages. This way at most one flash memory operation is necessary when accessing a variable. As the example shows, the elements on page 0 are accessed more frequently than those on page 1. Therefore, this page will probably be kept in RAM most of the time whereas page 1 can be replaced more often with other pages.

5.4. Evaluation

In this section we present evaluation results for *ViMem*. All experiments have been performed using Avrora. This simulator contains an energy model with detailed information about the energy consumption of the hardware components [Landsiedel et al. 2005].

5.4.1. Isolated Memory Access Performance

Table 5.2 compares the access speeds of a single variable access depending on where the variable is currently stored.

Table 5.2.: Typical latencies for different kinds of variable accesses

Type of access	**Delay**
Variable in RAM	1.09 µs
VM variable in RAM	18.72 µs
VM var. from buffer without page write	3.66 ms
VM var. from flash without page write	3.72 ms
VM var. from flash with page write to buffer	7.58 ms
VM var. from flash with page write to flash	19.83 ms

Variables that are not included in *ViMem*'s virtual memory system are always stored in RAM. The access to such a variable takes only 1.09 µs.

If a variable is stored in virtual memory, there is some overhead for each access even if the page containing the variable is already available in RAM. The reason for this is that *ViMem*'s runtime system has to check first if the page is currently stored in RAM. Since our hardware platform does not directly support virtual memory, this has to be implemented in software. Therefore, it takes up to 18.72 µs to access such a variable. This number also includes the address translation of the variable ID to the correct memory page and the offset within this page. However, in many cases this translation can be done offline by the compiler. Then the latency for variable accesses is reduced by 22 % to 14.65 µs (not shown in the table).

If a page is not stored in RAM but has to be retrieved from either the flash memory chip's buffers or the flash memory itself, the delay increases further. If the replaced page in RAM has not been modified and thus does not have to be written back to flash, it takes about 3.7 ms to read a new page from the flash buffers or the flash memory itself. These numbers also contain the processing overhead of *ViMem*'s runtime system that has to find a page to replace in RAM. However, the dominating factor is the transfer time of the page from the flash memory chip to the CPU via the Serial Peripheral Interface (SPI). This transfer time also prevails if a page from RAM has to be copied to one of the flash buffers. Because one page has to be read from the flash chip and another one has to be written to one of the buffers, the latency approximately doubles.

An even larger delay can be observed if another page has to be written from the flash buffer to flash memory in order to free a buffer for the new page coming from RAM. The whole cycle of writing that page to flash memory, writing another one from RAM to the flash buffer, and reading the new page takes more than 19 ms.

These numbers seem huge compared to a variable access in RAM. However, it should be noted that *ViMem*'s goal is to reduce the number of flash accesses, especially page writes. Therefore, in practice only a small number of variable accesses leads to such long delays, and most variables are accessed in RAM (see Sections 5.4.2 and 5.4.3 for details). In addition, since access to virtual memory is only allowed in non-time-critical functions, even long delays do not affect the operation of the application. Finally, in other domains, where virtual memory has been successfully used for years, similar delays can be observed. For example, the random access speed of typical hard disks for PCs is about 10 ms.

5.4.2. Application Performance

In this subsection we focus on the performance of complex applications using *ViMem*. We have modified two of the most RAM-intensive applications available in the TinyOS CVS repository [TinyOS]: TinyDB and Maté.

Experiment Setup

As already described in Chapter 4, TinyDB [Madden et al. 2005] provides an SQL-like query interface to the sensor network. With almost 30,000 lines of code it is one of the most complex applications available for TinyOS and as a part of the TASK system has been successfully used in real-world deployments [Tolle et al. 2005]. As described in Section 5.1, TinyDB requires the user to select at compile-time which functionality to include in the code image since the variables of all its components would not fit in RAM.

Maté [Levis et al. 2005] is a virtual machine that executes applications, which have been compiled to a special byte code format. Although this byte code representation is more compact than native code, RAM still is a limiting factor, since the user's application as well as its variables have to be stored there.

We have modified both applications to make use of *ViMem*. In TinyDB we have only put variables of application and routing components in virtual memory, as they are responsible for large amounts of memory consumption. Pointers to variables in virtual memory are still stored in RAM to avoid two flash accesses for a single data item. In addition, we have not added variables to virtual memory that are used in time-critical functions like "async events" or are cast to another type. TinyDB heavily uses the last kind of variables to implement a dynamic memory allocation mechanism. Although we have left this mechanism untouched, with significant changes to the application it would be possible to replace large parts of it with virtual memory.

Concerning Maté we have focused on the variables storing the application code capsules in RAM and left all other variables unchanged. First, these variables consume large amounts of RAM. Secondly, this part of Maté is essential to build more complex applications. Finally, Maté already provides mechanisms to adapt the size of these variables to the underlying hardware platform. If more memory for the byte code becomes available by using *ViMem*, it is easy to adjust Maté to take advantage of this.

Table 5.3.: Variables moved to virtual memory

Application	**Number**	**Size**	**Pages used**
TinyDB	75	569 bytes	2.15
Maté	1	792 bytes	3.0

Table 5.4.: Allocated space in RAM

Application	**Original**	***ViMem***
TinyDB	2,832 bytes	2,577 bytes
Maté	3,196 bytes	2,727 bytes

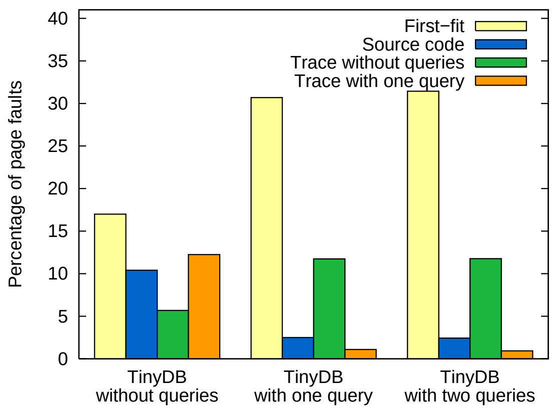
Table 5.3 summarizes the number and size of variables that are stored in virtual memory and shows the number of memory pages used. All of these modifications were done by simply adding the "@vm" and "@vmptr" attributes to variables or pointers, respectively. With Maté all the code capsules are stored in a single array. Therefore, just one variable has been marked with the "@vm" attribute. We have compiled both applications with their default settings.

Table 5.4 compares the size of allocated memory in RAM for the original and the *ViMem* versions of the applications. In addition to the size of the page currently kept in RAM the *ViMem* versions also include the RAM overhead of their runtime components as well as the flash memory access components from TinyOS (approximately 50 bytes).

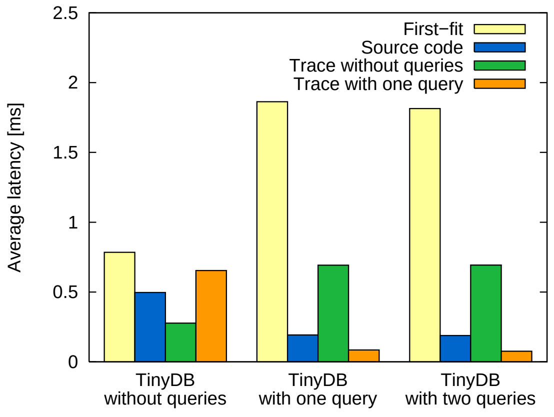
We have created several versions of TinyDB and Maté that use virtual memory and differ only in their respective memory layouts. The first version uses the same runtime components as *ViMem* but places data elements in the order in which they are processed by the compiler using a first-fit strategy. This approach makes use of the observation that variables declared together are often used together and, therefore, exploits the natural locality of the code. The other virtual memory versions use *ViMem*'s heuristic to create an optimized memory layout. One of them just relies on information about variable accesses from the source code whereas the other ones base their layout on simulation traces. For both applications one such trace has been created when no query or no byte code application, respectively, was running. For TinyDB the second trace has been created when one query to report the node ID was being executed. Similarly, for Maté we have run the CntToLeds application from the tutorial [Maté] for recording the second simulation trace.

All data access traces have been obtained from a deliberately simple setup with just five nodes in a grid topology. In contrast, the actual simulations of the applications have been done in a network of 50 randomly placed nodes. These changes in the simulation setup help to show that some differences between the scenario in which the access traces have been obtained and the actual operation environment do not influence the effectiveness of our heuristic.

For these experiments we keep just one memory page (264 bytes) in RAM. This is the worst case for a virtual memory system: if a variable from another page is accessed,



(a) Page fault rate



(b) Average access latency

Figure 5.9.: Simulation of TinyDB

this page has to be loaded from flash memory first. Having just one page in RAM eliminates side-effects of the page replacement algorithm. Therefore, the performance is just influenced by the memory layout.

We have simulated TinyDB and Maté using different scenarios. Each of them has been simulated for 1,000 seconds.

Simulation of TinyDB

Fig. 5.9 shows simulation results for the four variants of TinyDB described above in three different scenarios. The first two scenarios correspond to the ones used to get the data access traces. In the third scenario an additional query is dispatched which reports the values of the light sensors.

In Fig. 5.9(a) we show the ratio of variable accesses leading to a page fault, i.e., a read access from secondary storage (including reads from the flash chip's buffers). In many cases *ViMem*'s memory layout algorithm greatly reduces the number of accesses to flash memory. Using only the data references from the source code it is already able to decrease the percentage of accesses leading to page faults by at least 35 %. Nevertheless, memory layouts that have been optimized for a given scenario can reduce the percentage of page faults even further. For the scenarios executing queries in the best cases the page fault rate is just around 1 % (about 3,000 page reads). This excellent result is possible because only a subset of the variables in virtual memory is frequently used. In these experiments 90 % of all accesses refer to just 20 % of the bytes in virtual memory.

For the scenario when no query is running the page fault rate is slightly greater (approximately 5 % or 1,400 page reads) even if the memory layout has been optimized specifically for this scenario. The reason for this is that here a much smaller number of accesses is distributed over almost the same number of distinct data elements.

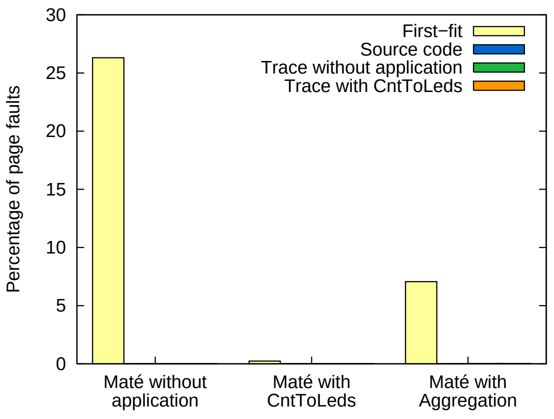
As the versions optimized with traces of the respective other situation show, the scenario used to obtain the access traces should not differ too much from the actual execution scenario. In fact, in such cases the version optimized using the source code can be even better than an application optimized for the wrong scenario – especially, if the simulation scenario was too simple for the execution environment. However, if more information about the execution scenario is available, using traces from simulation can reduce the number of page faults by about 60 % compared to just using the source code for optimizations.

With *ViMem* less than 7 % – and in the best case just 0.6 % – of all variable accesses lead to a page write to the flash buffer. In addition, at most 0.1 % of all variable accesses result in an actual write access to flash memory. The reason for this is both the efficient memory layout and the use of the flash chip's buffers as a second level of caching. In absolute numbers, in our simulations with *ViMem* each node usually performs just 2 such operations. Our wear leveling heuristic described in Section 5.2.3 ensures that write accesses are evenly spread across all allocated flash memory pages. The first-fit strategy, however, has to transfer a page to the flash buffer in up to 14 % of all variable accesses. Each node writes up to 1,696 pages to flash memory, although it uses the same buffering techniques as *ViMem*. We attribute this difference primarily to the distinction of pages that are mostly read and those that are written often.

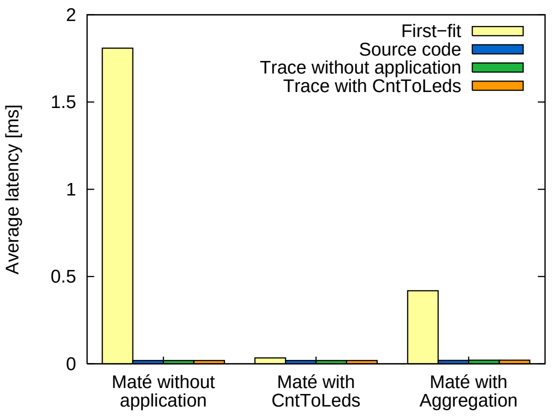
As the access latency directly depends on the number of page faults and write accesses, similar results can be reported for these measurements. As Fig. 5.9(b) shows, *ViMem* reduces the latency for TinyDB up to 95 % compared to the first-fit layout. However, even for the version optimized using the query trace, the average latency is still at least 75 μs whereas an access of a normal variable in RAM is performed in approximately 1 μs.

If more than one page is kept in RAM, most of these numbers will be further reduced, since a smaller number of flash accesses will be necessary. For instance, if two memory pages are kept in RAM, for the *ViMem* versions in many cases no page faults at all occur and the average access latency can be as small as 16 μs. Furthermore, in sensor networks processing power is usually not as constrained as other resources; in typical sensor network applications the CPU is idle most of the time. Therefore, we think that the overhead introduced by *ViMem* is acceptable for many applications.

We do not present detailed results for energy consumption because they are dominated by other hardware components and interactions between nodes. In addition, the timings of the TinyDB versions vary so that, for example, the applications send a different number of packets or turn on their LEDs for differing durations. Therefore, the values oscillate. In Section 5.4.3 we show results for energy consumption in a more controlled setting.



(a) Page fault rate



(b) Average access latency

Figure 5.10.: Simulation of Maté

Simulation of Maté

Our simulation scenarios for Maté follow the same pattern from TinyDB. First, we have simulated Maté without any (byte code) application running and then with the CntToLeds application, which correspond to the two scenarios used to get the variable access traces for *ViMem*. Then we have run the aggregation application from the Maté tutorial. This is a more complex application that builds a collection tree and calculates the average value of sensor readings in this tree. Like the third scenario for TinyDB this one shows the performance when the operation differs from the traces.

As Fig. 5.10 shows, for Maté many of the numbers are even smaller than for TinyDB. The reason for this is that in our scenarios only some of Maté's storage spaces for code capsules are used. Therefore, accesses typically refer to just a small set of variables that can be kept in RAM most of the time. This behavior, however, is typical for most applications that allocate their memory statically because they have to reserve enough space to deal with the worst case.

When no application is executed, there is only a small number of variable accesses (38 accesses per node, mostly for initialization). Therefore, the high page fault rate for the unoptimized version shown in Fig. 5.10(a) is somehow misleading as it is just 10 page faults in absolute numbers. The *ViMem* versions, however, do not show any page faults at all for this scenario as well as for the execution of CntToLeds. For this application even the unoptimized version achieves a page fault rate of just 0.23 %. This is because CntToLeds is very small and the code capsules used are (by coincidence) placed on a single memory page.

For the aggregation application the numbers remain excellent for all *ViMem* versions with approximately 6 page faults (less than 0.1 %). This application is still small enough to fit on a single memory page if the memory layout is chosen appropriately.

In fact, less than 6 % of the data elements allocated in virtual memory are used in 90 % of all accesses. This is because the total size of statically allocated memory has to be able to accommodate an application consisting of several large code segments. If some code capsules are only filled partially, the memory space in between is not used.

The short access latencies shown in Fig. 5.10(b) also reflect that a very small number of write accesses is necessary. If there are page faults at all, each node of the *ViMem* versions writes on average less than 4 pages to the flash buffers of which up to 3 are copied to the actual flash memory chip. The unoptimized version, however, has to transfer up to 699 pages to the flash buffers, which increases the latency.

5.4.3. Large Data Storage

In this subsection we evaluate *ViMem* with respect to two parameters: the number of memory pages in RAM and the total size of all data stored in virtual memory.

Experiment Setup

It is not possible to completely evaluate *ViMem* using TinyDB and Maté because in their current implementations these applications do not operate with several kilobytes more RAM than available. It would be interesting to see, however, how *ViMem* performs in this case. Moreover, it is also not possible to evaluate *ViMem* with simple applications, which, for instance, access memory sequentially or randomly. Results from such experiments would not be meaningful, either, because these accesses would not exhibit the patterns found in real applications.

Therefore, in order to evaluate the behavior of *ViMem* with larger data sets we have written a code generator that creates an application with an arbitrary number of data elements whose distribution of memory accesses is based on the accesses of a real application. A difference is, however, that for the generated application each data element has a size of 4 bytes ("uint32_t"). If the number of variables is greater than in the original application, the code generator adds some random jitter to avoid that always the same number of distinct variables is accessed. Therefore, when increasing the total size of data in virtual memory the number of variables actually accessed also grows.

For the experiments described in this subsection we have used this code generator to create applications that perform 5,000 data accesses, which are repeated 10 times. The basis for code generation was a data access trace from TinyDB where no queries were executed.

We have simulated these applications for 1,000 s. After completing the data accesses, the application switches the processor into power down mode, where it consumes less energy. Other hardware devices such as the radio or the LEDs have not been enabled.

In addition, no side effects because of other computations done by the application or interactions with neighboring nodes occur. Therefore, using this approach we can measure the pure overhead of the virtual memory system.

We show the results of two sets of experiments. In the first one we have created an application that allocates 3,168 bytes (12 pages) in virtual memory. We have then varied the number of memory pages that are kept in RAM. We intend to evaluate the overall performance of the system here which includes the overhead introduced by the replacement algorithm at runtime. In the second set of experiments we have increased the total size of virtual memory up to 15,840 bytes (60 memory pages) while keeping a constant number of pages in RAM. This is almost four times the size of RAM available on Mica2 sensor nodes. These experiments provide some insight about how *ViMem* performs when varying the data size.

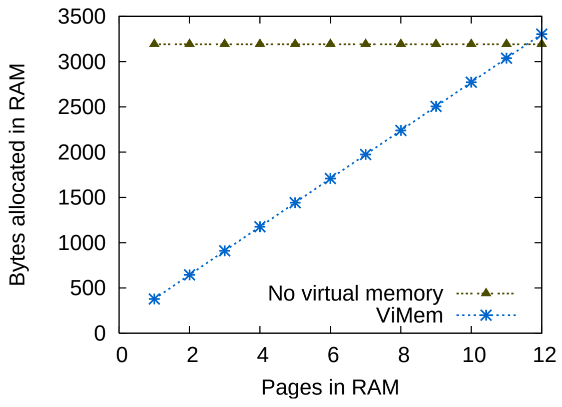
We have simulated the generated applications with different memory layouts. First, where possible, we have created a version that stores all data in RAM without using virtual memory. Obviously, such a variant could not be created for the second set of experiments because the total data size is larger than RAM there. Secondly, we have used a first-fit strategy to place data elements on virtual memory pages in the order in which they have been declared. The third version uses *ViMem*'s memory layout algorithm with information about variable accesses from the source code. Since in our generated application there are no branches or pointers, this variant has a perfect view of all data accesses, which in real applications could only be obtained through simulation. Finally, the last version uses the same trace but omits every fourth access when processing it. The idea of this variant is to see how *ViMem* performs if the operating environment differs from the simulation setup and the exact access trace is not known beforehand.

Number of Pages in RAM

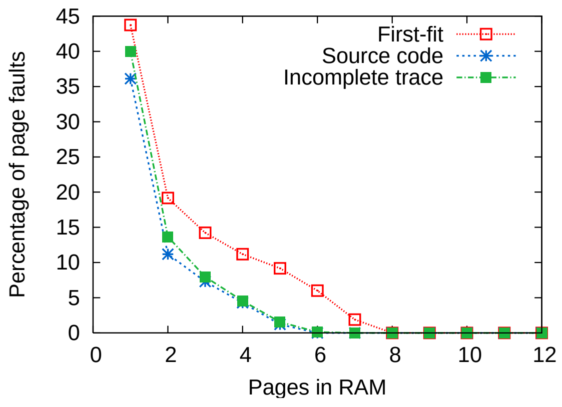
Fig. 5.11(a) shows the amount of RAM statically allocated by the application with and without virtual memory. If *ViMem* keeps just one page in RAM, it saves almost 90 % of RAM and allocates just 378 bytes (including all operating system components and the flash memory component). For each additional page the memory consumption increases by 266 bytes, which is just two bytes more than the size of the memory page itself. Only in the very last measurement *ViMem* allocates more RAM than the variant without virtual memory (overhead: 4 %).

Fig. 5.11(b) shows the percentage of variable accesses that lead to a page fault when the number of pages in RAM is varied. The results for one page are greater than those presented in Section 5.4.2 because the total data size is larger here. As expected, in these simulations the results using source code optimizations are the best ones. Here the memory layout algorithm can accurately predict the data access patterns.

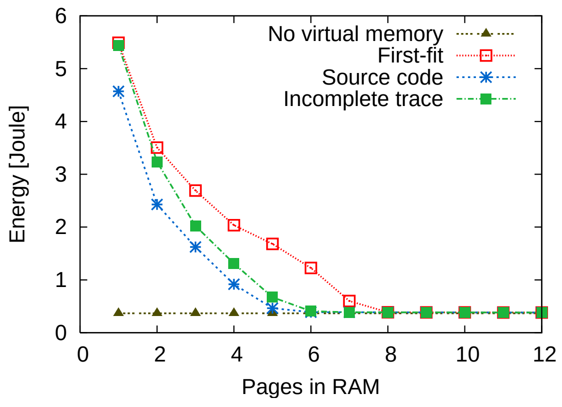
Of the 3,168 bytes allocated in virtual memory 1,676 bytes are actually accessed. Since



(a) RAM allocation



(b) Page fault rate



(c) Energy consumption

Figure 5.11.: Varying the number of pages in RAM

this size is much larger than just a few memory pages, conflicts between different pages in RAM occur with any memory layout for 1 and 2 pages in RAM. For these cases the optimized versions do not show much advantage compared to the first-fit approach. In fact, for the version using the incomplete access trace some more write accesses are necessary which further increases energy consumption (see Fig. 5.11(c)). If, however, with more pages such conflicts no longer occur, *ViMem* is able to create efficient memory layouts. For example, if at least 4 out of 12 memory pages are stored in RAM, the page fault rate for the optimized versions is less than 5 %. From 6 or 7 pages on, respectively, no more pages have to be read from flash. Compared to the unoptimized version *ViMem* avoids up to 3,995 page faults for the 50,000 variable accesses simulated.

Fig. 5.11(c) presents results for energy consumption. The values shown here just include the energy consumption of the CPU and the flash memory chip. The figure also includes the values for the application that stores all its data in RAM. Of course, this approach performs best with results that cannot be reached by any virtual memory solution. Nevertheless, the results for *ViMem* are encouraging. When no page faults occur, the energy overhead of the virtual memory solutions compared to the RAM-only version is just 0.02 J (5 % overhead). If accesses to flash memory are necessary, roughly half of the total energy is consumed by the CPU and half of it by the flash memory chip.

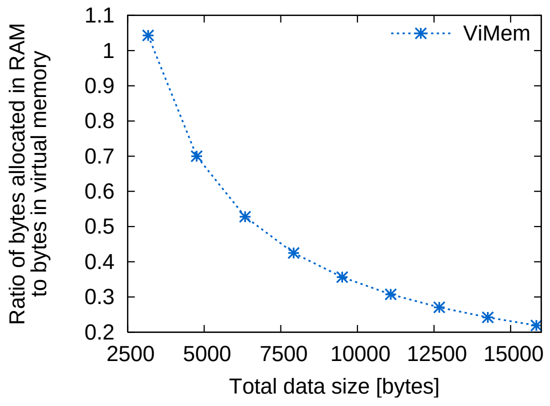
It should be noted that this application represents the worst case for *ViMem* when comparing it to implementations without virtual memory: Energy is only spent to access variables. No other devices such as the radio or the sensors consume energy and no additional computation is performed by the CPU. For example, if the radio is listening during simulation, energy consumption rises to 39.5 J for the application storing all its data in RAM. Therefore, for real-world applications, the overhead of *ViMem* is less significant, even if some accesses to flash memory are necessary.

Though not shown in the figure, the values for the access latency closely correspond to energy consumption as both of them depend on the number of read and write accesses to the flash memory chip. In the best case our heuristic's latency is just 4 % of the access latency of the first-fit strategy and only 15 μs worse than the version storing all its data in RAM.

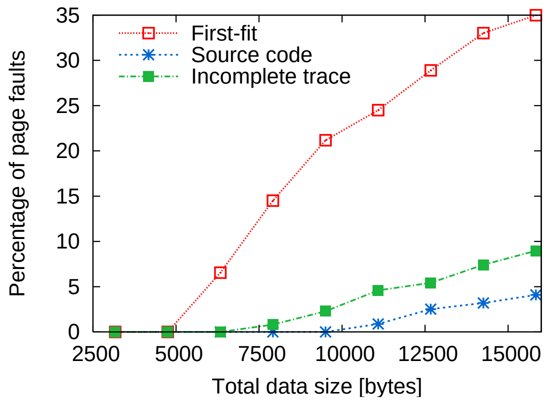
In summary, Fig. 5.11 suggests that with *ViMem* large amounts of RAM can be saved at justifiable overhead if more than one third of the total number of pages is placed in RAM.

Total Data Size

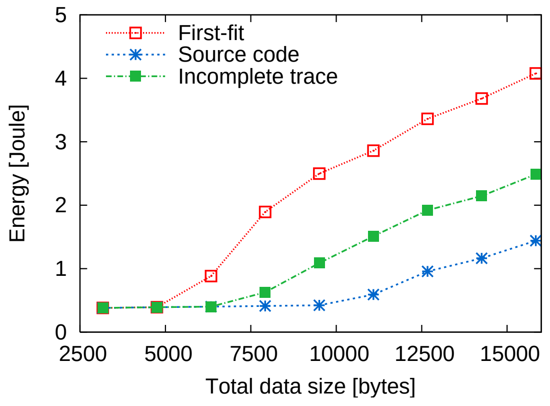
All the previously described applications could be implemented without virtual memory because they allocate less RAM than the 4 KB available on our implementation platform. With the following experiments we want to find out how *ViMem* performs



(a) Relative RAM allocation



(b) Page fault rate



(c) Energy consumption

Figure 5.12.: Varying the size of the data in virtual memory

if the size of the data is increased. Using our code generator we have created applications that allocate up to 15.5 KB of virtual memory. In all simulations the contents of 12 memory pages are kept in RAM.

Fig. 5.12(a) shows the ratio of bytes allocated in RAM (including all overhead and variables from other components used) to the size of virtual memory. As before, the only case where more memory is allocated in RAM than in virtual memory is the artificial one with all pages in RAM. In all other measurements virtual memory provides much more memory than it allocates in RAM. Each additional page in virtual memory has a RAM overhead of just 3.5 bytes (mostly for the data structures needed because of wear leveling) and provides 264 bytes to the application.

Fig. 5.12(b) presents the percentage of variable accesses leading to page faults. The numbers for the first-fit version increase to 35 % for a data size of 15,840 bytes (60 pages). For the versions of the applications whose memory layout has been created by *ViMem*'s algorithm, however, a much smaller number of page faults has been measured. For the best version the page fault rate stays smaller than 4.1 % in all cases.

Despite the randomness introduced when increasing the data size only a subset of all variables is accessed. This subset can be kept in RAM all the time if the total data size is less than 11 KB. Only after that the first page faults occur with the best *ViMem* variant. Again, in real applications such good results could only be obtained by using simulation traces for optimizations. If the incomplete data trace is used for optimization, the numbers increase faster but stay well below the first-fit approach. For this version the maximum page fault rate is 9 %.

Fig. 5.12(c) shows the energy consumed by the different versions of the application. With the best variant the results for the largest data size increase only by 1.1 J compared to the smallest size. For the other *ViMem* version energy consumption increases moderately by 2.2 J whereas the first-fit version with 60 memory pages consumes throughout the simulation 3.7 J more than the 12 page application.

Again, energy consumption is closely related to the average access latency. For this metric the maximum value of the first-fit approach is approximately 1.8 ms. Regarding *ViMem*, the latency increases only to 0.80 ms for the version optimized using the incomplete trace and 0.39 ms for the other one.

In summary, if sufficient information about access patterns is available, *ViMem* is able to significantly enlarge the memory available at only slightly increasing costs.

5.5. Related Work

In this section we present work related to *ViMem* that deals with virtual memory and data placement, uses flash memory for other purposes in sensor networks, or provides other solutions to the lack of memory on sensor nodes.

Virtual memory has a long history in operating systems research [Denning 1970] and is now a standard technique in modern operating systems [Silberschatz et al. 2002]. It allows the developer to use more RAM than physically available in the system by swapping out data to secondary storage. The system takes care of selecting the memory pages stored in RAM and translating addresses. Therefore, virtual memory is completely transparent to developers. Recently, t-kernel [Gu and Stankovic 2006] has applied this widely-used mechanism to sensor networks. However, this system does not modify the memory layout to minimize accesses to secondary storage, which is an important property of our approach. In addition, there the overhead of a page fault can occur with every variable access (even in time-critical functions) since the developer cannot select the variables that are to be placed in virtual memory. Finally, because it is based on load-time modifications of the code, this further increases the overhead at runtime.

For traditional computing systems some compiler techniques to optimize the memory layout for the properties of the memory hierarchy have been proposed. Most of them either use hardware support, simulation traces, or simply the source code to determine the memory layout. In many cases the main focus is on restructuring code, which is not needed for our scenario where the code itself is separate from (virtual) data memory. For example, Hatfield and Gerald [Hatfield and Gerald 1971] reorder code sectors to have parts close to each other if they are used together. Similarly, Hartley [Hartley 1988] duplicates code blocks to have functions always near their caller. However, these optimizations do not modify the arrangement of variables in memory.

Stamos [Stamos 1984] classifies different approaches to create a memory layout including both code and data. In his classification our approach would be a graph-theoretic algorithm that uses information from actual execution traces. However, he regards the detailed analysis done by our memory layout algorithm as infeasible for his own scenario, the placement of Smalltalk objects on virtual memory pages.

In addition, there are some techniques that target other parts in the memory hierarchy. For example, compile-time optimizations can be used to create an efficient memory layout concerning CPU cache misses. Although similar in spirit, the problem for CPU caches is different from the one addressed by our approach. Instead of placing data used together on the same memory page, Calder *et al.* [Calder et al. 1998] try to reduce cache misses by placing such entries in memory locations that are mapped on non-conflicting cache lines.

Muchnick [Muchnick 1997] gives an overview of techniques of compile-time optimizations for the memory hierarchy. In contrast to our approach, these techniques often involve modifications to the code such as loop transformations. Regarding the layout of data in memory, Gupta [Gupta 1991] proves that the problem of finding an optimal memory layout is NP-complete and describes a heuristic that uses information from the source code to arrange data in memory.

All of these optimization approaches are targeted to environments different from sensor networks. Therefore, they do not specifically address the properties of this domain (e.g., access characteristics of flash memory, energy considerations, etc.).

Sensor network applications and system components already use flash memory for different purposes. For instance, it is used to store sensor values directly on the sensor node [Tolle et al. 2005; Madden et al. 2005]. Storing the data in flash memory can be more energy-efficient than sending it to a base station [Dutta and Culler 2005]. In addition, code update mechanisms [Hui and Culler 2004; Koshy and Pandey 2005; Marrón et al. 2006a] use flash memory to store and process code updates before transferring them into program memory. Finally, ELF [Dai et al. 2004] is a file system for flash memory that allows application developers to store data without having to deal with the low-level properties of flash memory. As can be seen from this description, there is a variety of uses for flash memory in sensor networks.

Instead of using virtual memory, one could also build a sensor node which is already equipped with more RAM. For example, unlike most sensor nodes the BTnodes [Beutel et al. 2004] have 240 KB of additional RAM. However, there are currently no nodes available that are equipped both with more RAM and flash memory. If the advantages of flash memory such as non-volatile storage of sensor readings and its even larger size are needed, alternative solutions such as virtual memory have to be used. Since one of the central assumptions of sensor network research is that nodes are cheap, consume little energy, and have a small form factor, equipping them with more RAM in addition to flash memory or replacing flash memory with other, more expensive types of memory is not an option. In fact, in the future the class of inexpensive, energy-efficient nodes will continue to be equipped with very limited hardware resources [Gu and Stankovic 2006]. In addition, as described above, the experience from other domains shows that even large amounts of RAM do not lessen the need for virtual memory. Typically, more complex applications emerge if more memory is available.

5.6. Summary

In this chapter we have described and evaluated *ViMem*, our virtual memory system for TinyOS-based sensor nodes. *ViMem* does not require special hardware support for virtual memory and has been implemented for standard Mica2 nodes. We have identified the need for special memory optimizations in the domain of sensor networks and proposed a compiler-based heuristic to create an efficient memory layout. *ViMem* determines the layout based on data access traces obtained from simulation or the source code itself.

Since finding an optimal memory layout is an NP-complete problem, no (efficient) heuristic necessarily finds the best layout. However, as we have shown in Section 5.4, our approach can reduce the overhead of virtual memory significantly compared to approaches that just exploit the natural locality of variable declarations. Our simulations show that if the properties of the execution scenario are not known beforehand it is often better to just use the source code for optimizations instead of a simulation scenario that differs too much.

In spite of all optimization efforts, virtual memory introduces some overhead. As we show in the evaluation section, this overhead does not hinder the implementation of nontrivial applications for sensor networks. Compared to other energy consumers on the sensor node, the increase of energy consumption of *ViMem* can be almost neglected. In addition, as sensor network applications are typically inactive for long periods, no virtual memory accesses and, thus, no overhead occur during these sleep times.

From a developer's point of view, using *ViMem* is simple: Virtual memory variables just have to be tagged with a special attribute and can then be used as if they were in RAM. Therefore, the developer does not have to deal with the low-level details of flash memory accesses. In fact, the code expected by our pre-compiler is pure nesC code.

ViMem makes it possible to use virtual memory in the domain of sensor networks. Therefore, the memory limitations of sensor nodes are not as strict as before and fewer cross-layer interactions are necessary. Even if in future generations of sensor nodes more memory was available, the convenience of an optimizing virtual memory system would help to simplify the development of exciting applications, which are more complex than the ones we know today.

6. Abstractions and Algorithms for Energy-Aware Applications

This chapter presents the final part of the cross-layer framework: In order to address energy limitations (one of the major reasons for cross-layer interactions) explicitly, it introduces the abstraction of an energy level. This part of the framework focuses especially on those applications where the minimum required lifetime for all nodes is known in advance. At runtime the system selects one of the energy levels that provides good application quality while meeting the lifetime goals. Then the chapter shows how some local coordination can be used to better balance energy level assignments over time. It evaluates this part of the cross-layer framework using simulation as well as experiments with Mica2 nodes and presents an overview of related work.

6.1. Preliminaries

Traditionally, sensor network research has tried to maximize network lifetime, e.g., by exploiting redundancy or by applying cross-layer interactions. Although this is useful in many cases, for some applications the required lifetime is known in advance and there are no redundant nodes in their topologies. Therefore, instead of maximizing overall lifetime, it is more important that every single node stays alive for a user-defined lifetime and that during this time the application provides the best quality possible subject to the energy constraints present.

For example, in long-term structural health monitoring of bridges [Kim et al. 2007; Marrón et al. 2005c] the batteries of nodes can only be replaced every few years during regular inspections [Marrón et al. 2005c]. As the interval of these inspections is known beforehand, it corresponds to the lifetime goal of the network. Because of high energy costs for sensing and complex data analysis, measuring and analyzing sensor data is an energy-intensive task. In addition, due to the typically sparse network topology, a single node failure can partition the network and thus render large parts of it useless. Therefore, for this application it is more important to preserve network connectivity than to do complex analysis on every single node. For instance, if the battery capacity becomes scarce, nodes could use a less accurate but also less energy-intensive data analysis algorithm or – in the extreme case – switch to a reduced functionality mode, where they stop sampling and forward only analysis results of other nodes to the base station. In these cases, of course, such a node will no longer be able to offer its full functionality. Nevertheless, we argue that it is still more useful this way than if it stops working completely.

Even if network connectivity could be preserved with only a subset of the nodes, some applications require a high spatial resolution of sensors. If some nodes fail before the anticipated end of the experiment, this reduces the usefulness of the remaining data. For example, an application to monitor the microclimate of redwood trees [Tolle et al. 2005] transmits all data to a gateway node and stores it locally in flash memory to increase data yield. However, instead of executing both of these energy-expensive operations, the lifetime of a node could be extended if just one of them was executed. Then the node would still deliver some data even if some of it might be lost due to transmission errors, for example.

Similarly, for wildlife monitoring systems like ZebraNet [Liu et al. 2004] the user defines the duration of the experiment. In this application nodes gather GPS traces and forward their data in order to have other nodes (physically) transport it to the sink. If a node fails, no more data will be gathered for one of the animals. Here there are several possibilities to save energy by reducing the functionality. First, a node could no longer forward the data from other nodes and, therefore, decrease energy-intensive radio communication. Second, it could stop storing other nodes' data and avoid flash memory accesses. Finally, it could reduce energy consumption by decreasing the rate it queries its own position from the GPS receiver.

In these applications it is possible to identify parts which are more energy-intensive than others and which are not actually needed to provide some basic functionality. Therefore, the final part of our cross-layer framework is *Levels*, a novel abstraction for energy-aware programming of sensor networks that allows the developer to explicitly single out optional functionality [Lachenmann et al. 2007b].

With our approach developers can specify so-called energy levels in an application which differ in their energy consumption and the functionality they offer. The code within each such level is associated with the energy it consumes. At runtime *Levels* monitors the remaining battery capacity and the energy consumed in each level. It then selects an energy level that allows the application to achieve its target lifetime, if necessary with restricted functionality. This way the lifetime of individual nodes can be significantly extended and, for example, network connectivity can be preserved. Compared to manual implementations of such functionality this programming abstraction and its corresponding runtime system save much development effort. For example, the application developer no longer has to write code to estimate the energy remaining in the battery, the energy consumed by parts of the application, or the time full functionality can be provided.

Our approach is based on measuring the energy consumption of individual energy levels using an energy profiler with accurate simulation models [Titzer et al. 2005; Landsiedel et al. 2005]. At runtime each node tries to maximize the utility of the energy levels while achieving its lifetime goal. As we show in the evaluation, the abstraction of energy levels is useful in real-world applications and *Levels* is able to help meeting lifetime goals while providing good application quality.

However, if each node performs this optimization individually, all nodes will probably switch to similar energy levels at the same time. Then the overall application quality of the network – i.e., the average utility of all nodes – will be either excellent or poor but nothing in between. Therefore, we propose different approaches to better distribute energy level assignments over time. In particular, these approaches randomly assign the results from the local optimization or coordinate their assignments in a completely distributed way [Lachenmann et al.]. They increase the probability that, for example, there are always some nodes gathering data even if some have deactivated their sensors to meet their lifetime goal. Especially in dense networks like in the microclimate monitoring mentioned above such coordination can help to provide almost constant overall application quality.

These approaches cannot only be used to balance energy level assignments but can also be applied to activate and deactivate nodes if some of them are redundant. This is useful if the achievable network lifetime is extended with additional, redundant nodes. In this case it is sufficient if each of them is only active for a fraction of the overall network lifetime.

Deploying redundant nodes is possible in many applications. It is often easier than replacing batteries or deploying additional nodes later if the nodes are placed in inaccessible locations to do, for example, environmental monitoring. Frequently, the effects on application quality when activating or deactivating nodes are even greater than when switching energy levels.

Our solution has several benefits. First, the developer does not have to deal with the low-level issues of energy consumption, which simplifies the development of energy-aware applications. If the developer does not address these issues, there will probably be fewer cross-layer interactions and, therefore, their negative side-effects will be reduced. Second, our solution helps to ensure that a given lifetime is reached and that good application quality is offered. Third, with its distributed coordination our approach ensures that the overall application quality stays roughly constant. Fourth, the overhead for the developer is just small and we provide a powerful programming abstraction that allows for modular application development. Finally, the runtime overhead of our system is negligible.

6.2. Meeting Lifetime Goals with Energy Levels

In this section we present the abstraction of an energy level. First, we describe relevant design characteristics, the abstraction itself, and its integration into a programming language. Then we describe in more detail the corresponding runtime system including its battery model, its mechanisms to attribute energy consumption to an energy level, and the local optimization of energy levels.

6.2.1. System Design

Sensor Network Properties

Several properties of wireless sensor networks aid our approach of measuring energy consumption and switching between energy levels at runtime.

First of all, there is usually just a single application running on each sensor node. Therefore, the expected lifetime of a node depends only on one application that can be controlled more easily than a multitasking system.

Second, sensor networks typically exhibit some periodic behavior. For example, sensor readings are sampled periodically at user-defined time intervals. If the sensor network application reacts to external events, these events often repeat for sufficiently large periods. Thus it is possible to estimate future energy usage based on past consumption.

Third, because sensor nodes only have limited output capabilities and are often deployed in inaccessible locations, simulation has become an integral part of the development process [Titzer et al. 2005]. In addition, simulators are often equipped with detailed energy models [Landsiedel et al. 2005; Shnayder et al. 2004]. Therefore, similar to using simulation traces for *ViMem*, we can use simulation to get information about the energy consumption of a piece of software.

Fourth, most sensor nodes available today are equipped with voltage sensors. Since the voltage provided by a battery depends on its remaining capacity, we can use the voltage data to estimate the residual energy.

Finally, as the nodes are strictly energy-constrained, in the domain of sensor networks software developers are more concerned about energy consumption and node lifetime than developers in other areas. Therefore, we expect that most developers are willing to invest some effort for specifying energy levels and measuring energy consumption.

Design Goals

Our central design goal for *Levels* is to *provide a programming abstraction and runtime support that helps to meet the user's lifetime goals by deactivating parts of an application if necessary.* To achieve this main objective we have identified the following subgoals:

- The programming abstraction should allow for the definition of optional functionality and it should be general enough to support a wide range of applications.

- *Levels* should be easy to use and the development overhead should be limited.
- For a given lifetime goal *Levels* should provide good application quality, i.e., nodes should not live much longer than required.
- The system should have a low runtime overhead to avoid that the overhead absorbs its benefits.
- The runtime system should be able to deal with inaccurate energy estimations which are inevitable with inexpensive sensor nodes.

System Overview

Based on these design goals we have created the programming abstraction of "energy levels" that allows to specify optional code blocks. At runtime the system then decides which energy levels are active, i.e., which code blocks should be executed. This abstraction is described in more detail in Section 6.2.2.

Basically, our system is similar to well-known model predictive control (MPC) schemes [Camacho and Bordons 2004]: First, we build a model that helps to predict energy consumption by profiling the energy consumed by optional code (see Section 6.2.3). This model allows to compute the energy used by each level at runtime. Second, it is complemented by a battery model that maps voltage readings to the remaining energy usable by the sensor node (see Section 6.2.4). Third, using the information from energy profiling *Levels* keeps track of how much energy is consumed by each energy level at runtime (see Section 6.2.4). This part of *Levels* considers both energy that is consumed just once when executing a code block (e.g., to store some data in flash memory) and changes in the rate of continuously consumed energy (e.g., by enabling a sensor). Finally, together with the battery model this data allows to compute the expected node lifetime in each energy level. This information is then used to optimize the energy level for the remaining lifetime considering the given energy constraints (see Section 6.2.4). Like other MPC algorithms, our system considers just the result for the current time interval and later recomputes the remaining level assignments to better reflect the new situation.

6.2.2. Energy Levels

An energy level includes all statements that can be deactivated together to reduce energy consumption. Therefore, code in an energy level is optional for providing some basic functionality of an application. If a level is deactivated, however, the functionality of the application may be degraded. To put a code block into an energy level the developer has to place it into a conditional statement that checks if the level is currently active. The lowest energy level l_0 is always active and is declared implicitly; it includes all code that has not been added to any other level.

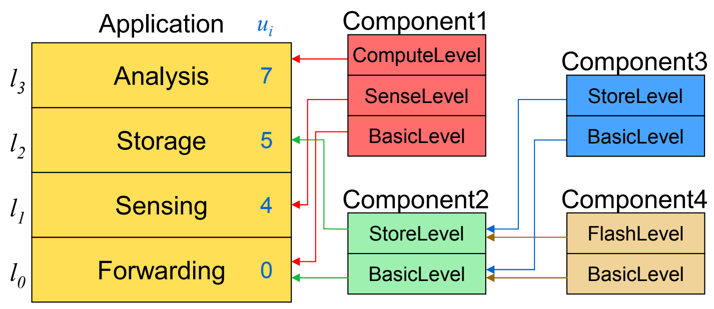


Figure 6.1.: Combining energy levels

All the energy levels form a stack where levels can be deactivated starting from the top one. If level l_i is active, all levels below it, i.e., $l_0, \dots, l_{i-1}$ are active, too. Therefore, the code in l_i can rely on the functionality of lower levels. Levels above l_i, however, might be deactivated.

Each energy level l_i is associated with a utility value u_i: The application developer can define this utility in a way that reflects the improvement in functionality.

Having a stack of energy levels does not mean that the functionality of an application has to increase monotonically with higher levels. By using appropriate conditions it is possible to, for example, transmit sensor readings in a low energy level to the base station whereas they are just stored locally (without being forwarded) in a higher level.

Levels assumes that higher levels lead to an increase in energy consumption. We expect this to be true for almost any application. Otherwise, energy levels should be merged because they are ill-defined. Such a situation could be easily detected during the development phase.

If an energy level is being activated or deactivated, the runtime system calls a special function to notify the application. It can use this function to adjust to the new level. For example, if all the sensor sampling code is extracted into an energy level, the application could turn the sensor hardware on or off in these functions.

Some applications might require that the system cannot change the currently active level while the node is, for example, sensing or analyzing data. *Levels* provides an interface for such applications that temporarily prevents level changes. To avoid that nodes stay too long in the wrong level, however, such periods should be relatively short. If a new level assignment is necessary during this time, it is applied immediately after the application allows level changes again.

The abstraction of energy levels nicely fits modular development in component-oriented languages like nesC [Gay et al. 2003]. If an application consists of several nesC components which define their own energy levels, it might be undesirable that each of them can be deactivated separately. For example, code in one component might depend on functionality of another one's (higher) levels. Therefore, using a "wiring"-like mechanism a component can combine levels of several components and thus ensure that they are only active at the same time. In Fig. 6.1 the arrows show this mechanism. For instance, it is used to combine the energy levels of Component3 and Component4. Likewise, the overall levels of the complete application can be created by combining the energy levels of its components. In addition, it is possible to insert levels from one component between those of another one. For example, in Fig. 6.1 StoreLevel of Component2 is mapped between the levels of Component1 in the application. However, it is not possible to change the order of the levels of a single component; doing so could break assumptions in the code.

Using this simple mechanism, which closely corresponds to nesC's wiring of interfaces, the overall energy levels of the application in the figure ($l_0, \ldots, l_3$) are formed. This application can deactivate functionality for data analysis, storage, and sensing if necessary. Forwarding functionality, however, is placed on the lowest level l_0, which is always active. The values for u_i in the figure refer to the user-defined utility of the energy levels.

By connecting all required levels to level l_0 the developer has full control of which energy levels have to be active for the current application. All other levels, however, can be deactivated if necessary. Therefore, *Levels* allows for the reuse of components with several energy levels, even if all of them are required to be always active for one specific application.

Syntax

Like the other parts of the framework, we integrated *Levels* into nesC [Gay et al. 2003]. Building upon this general-purpose language helps to achieve general applicability of our abstraction.

Fig. 6.2 shows a small code example of a component that provides two energy levels. The numbers in the declaration of energy levels determine their local order. However, these numbers do not have to be absolute or globally unique; other levels can still be inserted when wiring the component.

In this example each energy level consists of a single optional code block. If the highest level, "ComputeLevel", is active, the component performs some computation after receiving messages (line 10) and periodically reads a sensor value (line 16). If just "SenseLevel" is active, the component will continue sampling data but cease to do the computation. It would also have been possible to add else-branches here that run a less expensive computation task in lower levels, for example. In the implicitly declared default level l_0, neither "computeTask" nor "SensorADC.getData" will be invoked. Note that the code inside these two functions is regarded as a part of the energy level from which it is called.

Whenever a level is being activated or deactivated, one of the corresponding functions

```
 1 module Component1 {
 2   provides energylevel SenseLevel<1>;
 3   provides energylevel ComputeLevel<2>;
 4   ...
 5 }
 6 implementation {
 7   ...
 8   event TOS_MsgPtr ReceiveMsg.receive(...) {
 9     if (ComputeLevel.active) {
10       post computeTask();
11     }
12     return msg;
13   }
14   event result_t Timer.fired() {
15     if (SenseLevel.active) {
16       call SensorADC.getData();
17     }
18     return SUCCESS;
19   }
20   command void SenseLevel.activate() {
21     call SensorControl.start();
22   }
23   command void SenseLevel.deactivate() {
24     call SensorControl.stop();
25   }
26   ...
```

Figure 6.2.: Code example for a component specifying energy levels

("activate" and "deactivate") will be called. In the example the sensors are turned on just as long as they are needed (lines 21 and 24).

As the example in Fig. 6.2 shows, *Levels* requires only very small changes to existing nesC modules. Furthermore, wiring energy levels is completely analogous to wiring interfaces in nesC. This is shown in Fig. 6.3 and Fig. 6.4. Fig. 6.3 depicts the definition of Component2 from Fig. 6.1. Its optional energy level is simply created by combining the energy levels of Component3 and Component4. No wiring is needed for the lowest level, which is always present.

In Fig. 6.4 standard nesC syntax is used to wire the optional energy levels of the components from Fig. 6.1 to those of the application. Those energy levels are defined using the predefined levels "Main.EnergyLevel". Here the numbers in brackets define the utility of the different levels. Because the changes to nesC are minimal, we think that using energy levels should be easy for application developers.

For reference, Appendix A summarizes the changes to the nesC grammar needed for *Levels*.

```
configuration Component2 {
  provides energylevel StoreLevel<1>;
}
implementation {

  components Component3, Component4;

  StoreLevel = Component3.StoreLevel;
  StoreLevel = Component4.FlashLevel;

}
```

Figure 6.3.: Code example for wiring energy levels

```
configuration Application {
}
implementation {

  components Main, Component1, Component2;

  Main.EnergyLevel[7] -> Component1.ComputLevel;
  Main.EnergyLevel[5] -> Component2.StoreLevel;
  Main.EnergyLevel[4] -> Component1.SenseLevel;

  ...

}
```

Figure 6.4.: Code example for wiring the energy levels of the application

6.2.3. Energy Profiling

In order to correctly estimate the lifetime of the application, *Levels* has to know how much energy is consumed by each optional code block defined in energy levels. Getting this information on the sensor node itself is not possible since the energy consumed in individual blocks of code is too small to be accurately estimated using the node's built-in voltage sensor. Therefore, we make use of the fine-grained energy models available in simulators.

It should be noted that because of hardware differences the energy consumption of different nodes varies slightly [Landsiedel et al. 2005]. Currently, profiling can achieve only optimal results if the energy model of the simulator is calibrated to each node. Therefore, from our perspective an important design requirement for future sensor nodes is that they should be created from parts which show only little variations in energy consumption.

Compared to real measurements with instrumented sensor nodes and lab equipment, simulation has the advantage that the additional effort for the application developer is small. Furthermore, our approach allows to reuse code from unit testing to profile energy consumption. Therefore, in the following paragraphs we give a brief overview of the *nCUnit* testing framework before we describe our approach to energy profiling.

Unit Testing for Sensor Nodes

Unit testing using either tools like JUnit [JUnit] or custom test drivers has already proved to be a valuable technique in different domains including sensor networks. For example, the TinyOS distribution includes several small test applications for many operating system components. However, such custom test drivers often require significant development effort. In addition, developers usually have to manually check the results of the test cases. This is especially cumbersome for regression testing after modifying existing code.

Therefore, we have developed *nCUnit*, an automated unit testing framework. It assists the component developer when writing unit testing code by automating the setup of test cases and by checking the success of the tests with assertions. It reduces both the development effort for test cases and simplifies regression testing.

nCUnit is based on the Avrora simulator. Therefore, it directly runs the same code as Mica2 sensor nodes. Using simulation for unit testing has two major advantages. First, it allows to better control the environment and, thus, makes test results reproducible. Second, in the simulator details about the hardware state can be checked without influencing the runtime behavior of the component. This is possible since Avrora accurately emulates a sensor node's hardware.

The design of *nCUnit* takes into account the special properties of wireless sensor networks. Most important, the code of many typical TinyOS components does not modify some software state like, for example, in an application writing data to a database. The success of test cases for this code cannot be checked with conventional assertions. Therefore, besides functions for standard assertions that check, for instance, the value of a variable, *nCUnit* includes special assertions that allow to extend the simulator with custom code or that check if the function under test calls a given function of another component.

By extending the simulator the developer can check hardware properties that are unavailable to the code running on the sensor node itself or would significantly alter the runtime behavior. For example, such code could check if some data has been really written to flash memory or if a packet has actually been sent.

Checking if a function calls another function is useful because of the component-based structure of typical sensor network applications. As described in Chapter 2, in such nesC-based applications each component can call functions in any other one. To verify that the component under test actually sends a message via the routing component,

```
module TestAssertM {
  provides interface StdControl;
  uses interface Assert;
  uses interface StdControl as TestControl;
}
implementation {

  command result_t StdControl.init() {
    return SUCCESS;
  }

  void testFunction1() @test() {
    uint8_t* varPtr;
    // create Java class to check simulator assertion
    call Assert.assertJavaClass("example.TestJavaAssert", 1000);
    // check if function under test toggles the red LED
    call Assert.assertCalls("ToTestM.Leds.redToggle", NULL, NULL, COMP_NONE);
    // call function under test
    call TestControl.start();
  }

  void testFunction2() @test() {
    ...
  }
}
```

Figure 6.5.: Unit testing code for *nCUnit*

for example, the developer could replace that component and check when it is called. However, this would require additional effort to write such replacements and to connect them just for unit testing with the component under test. Therefore, *nCUnit* allows the developer to specify an assertion that checks if a given function is called.

To create a test suite with *nCUnit* an additional module with its test code is needed. For instance, Fig. 6.5 shows an example of such a module. Each function that contains a test case is marked with the "@test" attribute. This code is modified by *nCUnit*'s pre-compiler. It creates calls to the test functions and ensures that for each such function a separate simulation run is executed. Therefore, in each run the application is in a consistent state even if a previous test case has failed.

In the figure testFunction1() tests the function TestControl.start(). It uses two assertions. The first one uses the given Java class to do some checks from within the simulator whereas the second one checks if that function toggles one of the LEDs. The arguments of this assertion that are NULL in this example could be used to check if the given function is called with some specific parameter values.

```
module ProfilingM {
  provides interface ReceiveMsg;
  provides interface Timer;
  ...
}
implementation {
  ...
  void measureReceive()
      @energy("ReceiveMsg", "receive") {
    TOS_Msg msg;
    msg.addr = TOS_LOCAL_ADDRESS;
    ...
    signal ReceiveMsg.receive(&msg);
  }
  void measureTimer() @energy("Timer", "fired") {
    signal Timer.fired();
  }
  ...
```

Figure 6.6.: Test driver used for energy profiling

Measuring Energy Consumption with Simulation

Since *nCUnit* executes its test suites in a simulator, the approach we have applied there corresponds to our energy profiling technique. Therefore, the developer can reuse the test code from *nCUnit* for energy profiling. The only change needed is to tag all relevant functions in the test driver with an "@energy" attribute that tells our build system which functions should be used to measure energy consumption.

Fig. 6.6 shows example code that can be used to measure the energy consumption of the optional code blocks in Fig. 6.2. This module has to be wired to the component whose energy consumption should be measured instead of the components normally used to, for example, receive radio messages. Creating these simple functions and wiring the component correctly is the only additional effort needed from the developer. As already mentioned, similar or even the same functions can be used for unit testing.

A pre-compiler generates calls for all measurement functions tagged with the "@energy" attribute. The parameters of this attribute specify the name of the function that should be profiled. For each measurement function the energy profiler is executed several times, where – in order to avoid side-effects – each simulation calls only a single measurement function once. The profiler starts two separate simulation runs for all energy levels and each of their optional code blocks: one with a short duration t_1 and one with a longer duration t_2.

These energy measurements allow the system to compute two kinds of energy consumption for each of these code blocks: energy that is consumed once (i.e., when the code is executed) and energy that is consumed continuously (i.e., by changing the

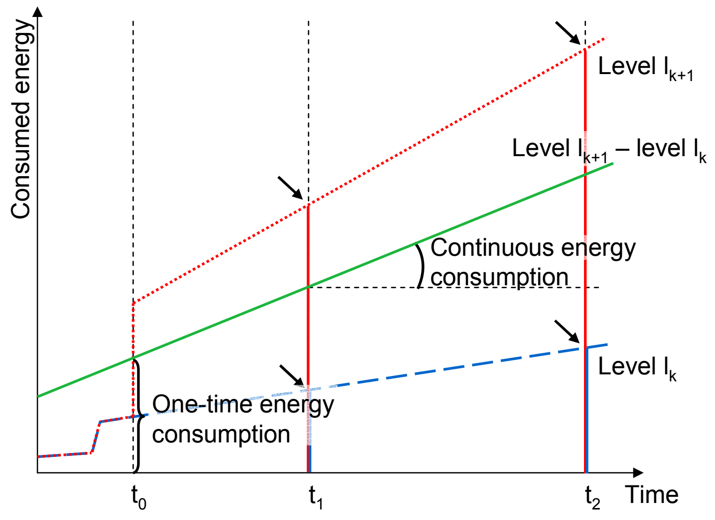


Figure 6.7.: Computing the energy consumption of a code block

state of a hardware device). For example, sending a message requires energy only once whereas turning on sensors leads to a change in continuous energy consumption. In addition, the measurements allow to remove the overhead introduced for setting up the test case.

Fig. 6.7 shows how this computation is done for the case when the function under measurement defines at most one optional code block for each of the energy levels involved. The function is called at the well-known point of time t_0. To get the increase in energy consumption of level l_{k+1} four measurements are necessary: the energy consumptions of the function under measurement in levels l_k and l_{k+1} at both t_1 and t_2 (see the arrows in the figure). Then the difference between the two levels is computed by subtracting their values. From the resulting points the slope of the energy difference, which corresponds to the change in continuous energy consumption, and the one-time energy overhead at t_0 can be computed.

This computation assumes that continuous energy consumption is linear. We expect this to be true on average for sufficiently long executions. For example, if a timer is activated to periodically invoke some code or if the sensor board is turned on, the average energy consumed will increase linearly with time.

Such a computation is done for each optional code block of all energy levels. If there are several such blocks in the function under measurement, in each run an additional block (from the beginning of a function to its end) is activated. Using the same principle as outlined above now for each code block, the difference in the energy consumed can be computed. Activating code blocks incrementally allows to measure their individual energy consumption while still ensuring that they can rely on the code in preceding blocks to be active, too. At runtime, of course, in the actual application all blocks of an energy level are active at the same time.

After executing all simulations, the energy profiler analyzes its log files and performs the computation outlined above. It then stores the energy consumption of each code block in a central file. This file is later read when compiling the actual application to insert energy values into the code.

Our profiling approach has several advantages. First, reusing unit testing code ensures that the code is executed in a controlled setting where, for example, messages from other nodes, unexpected sensor readings, or interactions with other components do not alter the application flow. Second, these measurements do not only include the energy spent by the CPU to run the code under test but also the energy consumption of other hardware like the radio or flash memory chips. Finally, unlike existing energy profilers [Landsiedel et al. 2005], which map energy consumption to code blocks, or systems monitoring the energy state of hardware components at runtime [Dunkels et al. 2007; Jiang et al. 2007a] our approach allows to include the energy consumption of asynchronously executed code (e.g., TinyOS tasks, timers, split-phase events) in the measurement. This is important to get the total energy consumption originating from the code block: Since this code is executed from a certain energy level, its energy consumption has to be attributed to that level.

Special Cases

There are two special cases to consider: the energy consumed by the lowest level l_0 and energy consumption that depends on some state of the hardware or software.

First, we do not measure the energy consumed by the default energy level l_0 and rather compute this value at runtime by subtracting the energy of all other levels from the overall energy consumed. This decision helps to keep the runtime overhead of *Levels* small since this level would be present in every single function. Furthermore, because there are no optional code blocks in level l_0, profiling could be done only at a coarse granularity and, therefore, would be probably inaccurate.

Second, for some code the energy consumption can differ depending on the state of the hardware and the application. For example, if – within an optional code block – the application tries to turn on a hardware device that is already enabled, executing this code will not change energy consumption. To address this issue the application developer can provide a condition in the "@energy" attribute that will later be checked at runtime. Therefore, each measurement function can refer to a different state of the component. For example, the condition could check an already existing state machine or read out the status of a hardware device. The system stores separate information about energy consumption for each condition. Depending on which condition applies, *Levels* attributes the correct energy consumption to the code block at runtime. However, the developer should choose these conditions in a way that allows to evaluate them efficiently. Otherwise, the overhead for checking the condition at runtime could outweigh the benefits.

If a level is not active at runtime, evaluating such conditions can be difficult since the deactivated code might have some effects on them. For example, if a function adds sensor readings to a buffer before storing all of them together to flash memory, only one of several calls will result in an energy-expensive write access to the flash chip. To deal with this problem our energy profiler calculates the probability that a code block is called with one of the conditions defined and computes the average continuous energy consumption of each level. For this purpose we have to run the complete application in a realistic scenario rather than execute unit test code. This can be done while testing the whole application. In these measurements we rely on the previously profiled energy consumption of code blocks; we just count how often they are called for each of the conditions given by the application developer. Since the information obtained from simulating the complete application is less accurate than the atomic energy measurements of code blocks, we use it only when necessary, i.e., for code blocks of deactivated energy levels.

6.2.4. Runtime System

In this section we describe *Levels*' runtime system. The runtime system has three tasks: estimating the remaining energy from voltage readings, attributing energy consumption to energy levels, and adjusting the active levels at runtime.

Battery Model

To estimate the remaining lifetime it is necessary to know at runtime how much energy is left in the battery. For this purpose we have built a simple battery model that maps voltage values to the remaining usable battery capacity.

Creating such a model has not been the main focus of our research. In fact, if there is a hardware power meter like SPOT [Jiang et al. 2007a] that accumulates the current drawn from the battery, better accuracy could be obtained than with this model. However, since the small, inexpensive devices we target have only voltage sensors, battery voltage has to be mapped to the remaining energy. To do this mapping we created a battery model for the specific type of alkaline-manganese dioxide batteries [Duracell Batteries] that we use in our experiments.

We opted for a simple but efficient model: for each distinct voltage reading we store the average remaining energy of this value in program memory. The overhead of this approach is minimal since it does not require any computation at runtime. Nevertheless, *Levels* is flexible enough to be combined with more advanced (but possibly more complex) battery models.

Because typical sensor network platforms like Mica2 nodes are not equipped with a voltage boost converter [Polastre et al. 2005b], the current draw I depends linearly

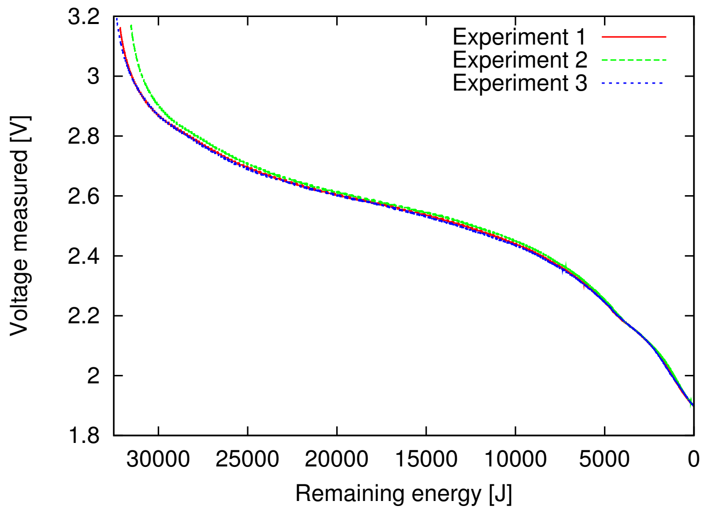


Figure 6.8.: Battery discharge characteristics from three experiments

on the battery voltage U. Thus the resistance R remains constant. From $E = U \cdot I \cdot t$ and $R = \frac{U}{I}$ the effect on energy (and power) is quadratic: $E = \frac{U^2}{R} \cdot t$. However, when creating our battery model we assumed a constant voltage $U_{const} = 3\,V$ for the computations. Therefore, instead of mapping the actual energy E to the voltage readings, our battery model and all energy values in the rest of this chapter refer to values for $E \cdot \frac{U^2_{const}}{U^2} = \frac{U^2_{const}}{R} \cdot t$. This simplifies computations at runtime greatly because the energy consumption of a code block can be assumed to be independent of the current supply voltage of the sensor node. Nevertheless, using the same approach in the creation of the model and at runtime leads to consistent results that allow for accurate computations.

To create the battery model we built an application for Mica2 nodes that periodically measures the voltage and transmits this data via radio until the batteries are drained. Using the energy model of the Avrora simulator [Landsiedel et al. 2005] we later computed the total energy consumed by this measurement application throughout the node's lifetime. The result of this computation is not necessarily the total energy available in the battery but rather the energy that is actually usable by the sensor node. For our purposes this number is more relevant because this is also the energy available at runtime. Ignoring for simplicity effects like the influence of temperature on the batteries, this data allows to relate voltage measurements with the usable energy left.

We performed several experiments with this application and created the model used at runtime by computing the average voltage reading for each energy value. Fig. 6.8 shows the discharge behavior of three batteries. Although there are some differences, the curves are almost equal when the batteries are almost empty. Particularly there a good energy estimation is important to accurately meet a lifetime goal.

Since the relationship between voltage and the remaining energy is not linear, the differences in energy values between two consecutive voltage readings can vary significantly (see Fig. 6.8). This directly affects the accuracy of the mapping. For example, in our battery models the differences vary between 7 J and 412 J for a battery with about 32,000 J usable capacity. Similar differences can exist between the models of several batteries, especially close to their total capacity. We make this expected error available at runtime. This makes it possible to defer computations until significantly more energy than that has been consumed; hence the influence of the inaccuracies is reduced.

Attributing Energy Consumption to Energy Levels

The runtime system is responsible for attributing energy consumption to energy levels. First, it is called whenever an optional code block is about to be executed. It then checks if the energy level is active and adds the energy consumption of this code block to the total energy consumed by the corresponding level. Second, periodically (every few seconds) it adds up the energy that has been consumed continuously in the current interval. Finally, periodically (every few hours) it uses this information and computes the optimal level assignment for the time remaining.

If an optional code block of an energy level is about to be executed, the system checks if the level is active. Only if it is active, the code will be executed. Furthermore, the runtime system uses the data obtained with energy profiling (see Section 6.2.3) and adds the energy consumption of the current block to the overall energy consumed by the level. If a code block of level l_i is reached by executing code belonging to level l_j with $j > i$, the system correctly attributes the energy consumed by this code to level l_j. In addition, it updates continuous energy consumption if it is changed by the code.

The same information is also updated for blocks belonging to the next higher energy level in the stack, which is actually not executed. This way the system can predict the energy consumption after increasing the current level. However, we do not monitor the energy consumption of even higher levels because it is unclear which of their code will be additionally reached if the levels in between are activated. For example, if currently only level l_i is active, the system cannot tell whether or not the application will reach more code blocks of level l_{i+2} from the code in level l_{i+1}. Therefore, energy consumption of l_{i+2} would not be accurately predicted.

Keeping only information up to the next higher level also restricts the levels that can be selected when adjusting the current level. Therefore, in each adjustment the system can increase the current level by at most one. This problem does not occur with lower levels because their energy consumption can always be accurately predicted. Thus several levels can be skipped when switching to a lower level.

As already mentioned, for each energy level, *Levels* keeps information about its continuously consumed energy, e.g., for a hardware component that has been enabled in an energy level. The runtime system periodically adds the energy continuously consumed

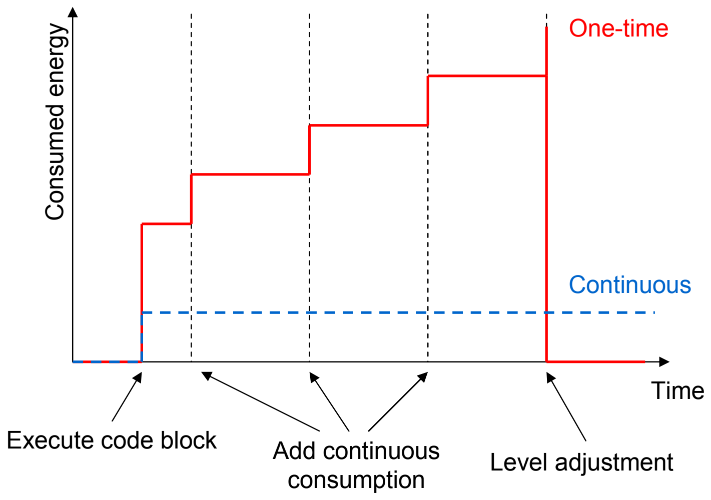


Figure 6.9.: Accumulating the energy consumed by a level

in the last few seconds to the one-time energy consumption of the code. This approach provides finer granularity and, therefore, better accuracy than doing this only when computing the energy level assignment. In addition, it minimizes overhead because it requires less state and computational resources compared to calculating this data whenever continuous energy consumption changes.

Fig. 6.9 summarizes how the runtime system computes the energy consumed by an optional code block. When the code block is executed, both one-time and continuous energy consumption are updated. The system then periodically adds continuous energy consumption to the energy consumed by the level. After some time this energy consumption is reset when computing a new level assignment.

Adjustment of Active Energy Levels

Levels uses the information about the energy consumption of energy levels to periodically adjust the currently active level. In each adjustment it tries to maximize the utility of the energy levels for the time remaining while meeting the lifetime goals. Formally this corresponds to the following optimization problem. Given the current lifetime t, the total required lifetime T_{req}, the remaining energy E_{rem}, and the energy levels $l_0, \ldots, l_{n-1}$, which have the utility values $u_0, \ldots, u_{n-1}$ and consume $P_0, \ldots, P_{n-1}$ energy units per time interval, find the durations $t_0, \ldots, t_{n-1}$ that maximize the utility of the energy levels for the remaining lifetime $T_{req} - t$:

$$\begin{aligned}
\text{maximize} \quad & \sum_{i=0}^{n-1} u_i \cdot t_i \\
\text{subject to} \quad & \sum_{i=0}^{n-1} t_i = T_{req} - t \\
& \sum_{i=0}^{n-1} P_i \cdot t_i \leq E_{rem} \\
& t_0, \ldots, t_{n-1} \geq 0
\end{aligned}$$

The first equation formalizes the maximization of the utility over time. Each t_i corresponds to the duration for which Level l_i is the highest active level. The constraints then specify that the still needed lifetime has to be met and that enough energy has to be available. Using a linear equation for the energy constraint is only possible because our battery model returns the energy for a (hypothetic) constant voltage instead of actual energy values. Finally, the last equation excludes solutions with negative time durations.

The optimization problem can be solved using well-known algorithms from linear programming [Chvátal 1983]. In our implementation we use the Simplex algorithm, which is the standard method to solve such problems. Since our implementation uses efficient fixed point arithmetic, the computational overhead of this algorithm is small. Furthermore, we limited the overhead by defining a maximum number of iterations after which the algorithm aborts even if it has not found the optimal solution yet. This limit is reached only seldom in practice, however. As we show in the evaluation, the computational overhead can almost be neglected even on resource-constrained sensor nodes (see Section 6.4.4).

The results of the optimization are the t_i values that specify for how long each energy level should be the highest active one. Because there are just two constraints (in addition to the exclusion of negative values), the Simplex algorithm always returns a result with at most two non-zero time intervals. Therefore, independent from the total number of energy levels at most two of them will be activated. We leverage this property in our distributed algorithm (see Section 6.3.2).

The system tries to compute a new level assignment periodically with a low frequency (e.g., every two hours). Repeating this computation is necessary since energy load may vary over time and because of possible inaccuracies in previous adjustments. However, due to the discharge characteristics of batteries the inaccuracies of the measurement might exceed the actual energy consumed, especially for low-power applications. In this case it is not possible to compute meaningful results. Therefore, we use the expected accuracy from the battery model at runtime: only if the energy consumed by code in energy levels is sufficiently big, the algorithm tries to compute a solution. Otherwise, it waits until the next measurement. Although this reduces the agility of the system, it helps to obtain correct results. In addition, to further reduce the fluctuations because of inaccurate measurements, we use a moving average to smooth the energy values used for computation.

Moreover, to deal with inaccurately estimated remaining energy and with possibly varying load within an energy level, *Levels* adds a safety factor to the lifetime still required in order to make sure that the node can meet its lifetime goal. This design decision leads to the side effect that the average level achieved is slightly below the optimum because the lifetime goal is usually exceeded. We opted for this conservative policy to ensure that no node runs out of energy early. Furthermore, as the safety factor depends on the remaining lifetime required, this issue is addressed by periodically recomputing the level assignment. The node will – in later computation rounds – switch back to higher levels if sufficient energy is still available.

To minimize the complexity of the runtime system, *Levels* does not change levels between these computations, even though the Simplex algorithm returns a complete level assignment for the remaining time; we just switch to the highest level of the result for optimal application quality. However, depending on the energy consumption of the application and the current accuracy of the battery model, several tries might be needed until the level assignment can be recomputed. Therefore, a level is selected only if the algorithm expects to be executed again before the computed time duration. Otherwise, *Levels* already switches to the next lower level of the solution.

Energy Level Assignment over Time

The level assignment described so far assigns energy levels independently on each node. Therefore, if the load on the nodes is roughly equal, all nodes in the network will change energy levels almost synchronously. Over time the overall application quality of the network will be very high in the beginning before suddenly becoming very low. Depending on the application this behavior might not be anticipated by the user.

One way to address this problem is to define the lowest energy level such that it is still useful to the application (e.g., nodes still sample data with lower-power sensors). Alternatively, the user can define the target lifetime in a way that it is actually achievable by sufficient nodes with their full functionality (e.g., at least by the leaves of the routing tree).

If such a definition of energy levels or lifetime requirements is not possible, distributing energy level assignments better over time often requires application-specific knowledge. For example, in some applications neighboring nodes should be fully active at the same time because they cooperate whereas in other applications nodes in high energy levels should be distributed uniformly throughout the network. Therefore, this problem can be addressed best by the application. Nevertheless, *Levels* provides supporting mechanisms: The application can give hints which level to select and *Levels* can introduce some randomness. Finally, as we describe in Section 6.3, *Levels* provides a distributed algorithm that coordinates nodes such that the average energy level remains roughly constant.

With the first mechanism the application performs its own coordination among nodes. It does not, however, directly assign an energy level because, otherwise, the lifetime goal could possibly not be met. Instead, it tells *Levels* which energy level from the result of the optimization problem to select first: the low or the high one. Over its lifetime each node will still use all energy levels from the results. Therefore, this approach cannot guarantee that really all nodes selected by the application operate in one specific level. However, it ensures that the lifetime goal can actually be met.

The second mechanism is similar but – instead of using information from the application – with this approach each node randomly decides which level from the local optimization result to select first. If the nodes have similar results from their local optimization, this approach distributes level assignments uniformly in the network. Although it might not provide optimal results, this mechanism is well-suited for low-power applications since it does not require any coordination among nodes. However, as we show in the evaluation, with minimal overhead our distributed algorithm is able to perform even better results.

6.2.5. Integration within the Framework and with TinyCubus

This section outlines how *Levels* has been integrated with other parts of the cross-layer framework and with TinyCubus.

Like the other parts of the cross-layer framework *Levels* has been developed in a way that allows to use it individually without requiring the rest of the framework. Nevertheless, it can be integrated seamlessly.

For using *Levels* together with *TinyXXL* no changes are necessary. If code accessing data in the *TinyStateRepository* is placed in an energy level, the access costs for variables in the *TinyStateRepository* are automatically considered when running energy profiling. The only change done was to create two energy levels for *Neidas*, the neighborhood data sharing algorithm. The first level includes all the code for sending requested data whereas the second one contains the code to send requests. Per default these energy levels are mapped to level l_0 and, therefore, are always active. However, the application developer can define them to be optional by changing some constants.

If *ViMem* is used, the energy consumption of a memory access may vary if the corresponding page is currently available in RAM or if it has to be retrieved from flash memory first. When a memory page has to be transferred from flash memory, energy consumption can increase even more if another page has to be written back to flash memory.

Like *Neidas*, we instrumented the implementation of *ViMem* with energy level code. This allows us to profile the energy consumption of operations like loading and storing flash memory pages. Since these operations are not optional, i.e., the system would no longer work correctly if their code was not executed, this energy level is mapped to l_0 which is always active. This code block has just been introduced to accurately attribute energy consumption if an access to virtual memory is done from a code block in another level. Therefore, the application developer does not have to create separate energy measurements for the cases when variables are already in RAM and when they have to be loaded from flash memory. This solution assumes, however, that over time always the same percentage of memory accesses will lead to a page fault.

The integration of *Levels* with TinyCubus is more difficult. In fact, integrating *Levels* completely in TinyCubus is not possible since some of its functionality is complementary to TinyCubus. As described in Chapter 2, the TinyCubus framework adapts applications based on their requirements, system parameters, and optimization parameters. One of these parameters can be energy consumption, which is also monitored and adjusted by *Levels*. The adaptation of TinyCubus is different from the adjustments of energy levels: TinyCubus allows to parameterize components and to replace them at runtime with different implementations.

If TinyCubus performs its adaptation, *Levels* could no longer predict the future energy consumption and the lifetime of the nodes accurately. Nevertheless, parts of *Levels* have already been created with the intention of integrating them in TinyCubus.

First of all, changing an energy level of a component corresponds to parameterizing this component. Therefore, the abstraction of *Levels* can be mapped to the mechanisms of TinyCubus. However, significant changes to the runtime system of *Levels* are necessary if it should take into account additional properties like reliability or the delivery ratio instead of just looking at energy consumption.

Second, there has to be a way for TinyCubus to estimate the remaining energy. For this purpose it can use the battery model of *Levels* that maps voltage readings to energy values.

Finally, TinyCubus has to know how much energy some adaptable code consumes. This functionality can build upon the simulation-based energy profiling of levels and the corresponding runtime system that keeps track of the energy consumed by different parts of the application. However, it should not just measure and track the energy consumption of the code in energy levels but probably also that of additional code. Since a complete implementation of the adaptation mechanism used in TinyCubus is not available yet, the details of this integration of *Levels* have to be left for future work.

6.3. Distributed Assignment of Energy Levels

The basic approach described so far might lead to large fluctuations in application quality of the network since each node optimizes its energy level assignment without taking into account those of its neighbors. To better balance the utility of energy level assignments we have provided an interface for the application to influence which nodes select what level. However, many applications do not need this flexibility because they are expected to require a roughly uniform distribution of utility values in the network. Thus such functionality should be part of *Levels*. For this purpose we have introduced our randomization scheme in Section 6.2.4. This mechanism operates purely locally and, therefore, does not add any overhead for communication. However, coordination among neighboring nodes is likely to yield even better results than the randomization mechanism at an acceptable overhead. Therefore, this section presents a distributed mechanism to better balance energy level assignments over time.

In many applications redundant nodes can be added to increase network lifetime and the resilience to node failures. If these nodes can be deactivated for some time, they have the same energy available for fewer active time intervals. This way higher energy levels can be selected and the application quality can be increased further. Likewise, network lifetimes that are longer than the maximum lifetime of each node become possible. Although we do not model deactivated nodes as another energy level, the same mechanisms can be used to balance energy level assignments and to determine when to activate redundant nodes.

Determining from all nodes the set of active ones is closely related to existing coverage and topology control algorithms. However, we only deal with the problems addressed by these algorithms implicitly through distributing active nodes in the network and controlling network density. Unlike our approach, the vast majority of coverage and topology control algorithms tries to *maximize* network lifetime [Cardei and Wu 2006; Huang et al. 2006; Cerpa and Estrin 2002; Ye et al. 2003]. For *Levels* a different optimization goal is needed because we assume that the required lifetime is given by the developer.

6.3.1. Problem Description

This subsection describes the problem of finding an optimal schedule to activate nodes and assigning their energy levels. Basically, this problem corresponds to finding a permutation of the active time intervals and energy level assignments where the deviations from the average over time are minimized. The nodes still try to provide the best quality possible for a user-defined network lifetime rather than maximizing that time.

All our assumptions with the exception of the presence of redundancy are also valid here. For example, we continue to assume that the developer specifies the overall required lifetime for the complete network. In the following equations, we refer to this lifetime value as $\overline{T}_{req}$.

Furthermore, we assume that – besides the required lifetime for the whole network – the individual required lifetime T_{req} is known for each node. This lifetime can either be directly specified by the user or computed from the number of neighboring nodes if the user defines the desired number of active nodes in an area instead.

In addition, our algorithm assumes that nodes are not mobile. This is necessary since it computes the schedule for a longer time based on information about the nodes present in the vicinity. This would not be possible if the nodes moved. Furthermore, some (coarse-grained) synchronization among nodes is needed because nodes have to transmit their results to their neighbors. If a node is deactivated, it has to wake up to receive these messages from its neighbors.

Using the total required lifetime of the network $\overline{T}_{req}$ and the time $\overline{t}$ that the network has already been operating, we define the remaining required lifetime of the network $\overline{T}_{rem}$:

$$\overline{T}_{rem} := \overline{T}_{req} - \overline{t}$$

Similarly, T_{rem} can be defined for each node as the required lifetime still remaining:

$$T_{rem} := T_{req} - t$$

If the remaining lifetime of node j is used, this is expressed as $T_{rem}(j)$. This function is defined for each node with $1 \leq j \leq R$, where R is the number of nodes to consider (e.g., the nodes in the radio neighborhood, including the current node).

Based on the lifetime of the nodes a schedule can be computed to activate and deactivate them over time. For this purpose we use an approach that tries to keep the number of active nodes constant, i.e., close to the average over time. Furthermore, as a secondary goal, not just the number of active nodes should stay close to its average, but also the utility of the currently selected energy levels. Determining the set of active nodes is the primary goal since we assume that variations in energy level assignments are less critical for overall application quality than variations in the number of active nodes. The input for balancing energy levels is the result of each node's local maximization of the utility since each node should still provide the highest utility (i.e., the best application quality) for its required lifetime. Only if both the number of active nodes and the average utility value are (roughly) constant, the application quality will not vary over time.

We do not interpret deactivating a node as an additional (lower) energy level because such a solution would not correspond to the semantics of energy levels that we have introduced in Section 6.2.2. Therefore, applications could not be easily adjusted if coordination among nodes was added.

1:	Node 1: 011	**4:**	Node 1: 101	**7:**	Node 1: 110
	Node 2: 001		Node 2: 001		Node 2: 001
2:	Node 1: 011	**5:**	Node 1: 101	**8:**	Node 1: 110
	Node 2: 010		Node 2: 010		Node 2: 010
3:	Node 1: 011	**6:**	Node 1: 101	**9:**	Node 1: 110
	Node 2: 100		Node 2: 100		Node 2: 100

Figure 6.10.: Combinations of activation schedules for two nodes

Computing a Schedule for Activating Nodes

To compute an optimal schedule for activating nodes we express the schedule of each node as a string of 0 and 1. Each character corresponds to a time interval. If it is 0, the node is turned off in this interval whereas it is turned on if the corresponding character is 1. The length of the string corresponds to the total remaining network lifetime $\overline{T}_{rem}$ and the number of 1 characters is the remaining lifetime of the node (T_{rem}). A valid schedule for the node is, therefore, a permutation of this string. Such a string is given for each node. A valid schedule for the network is a combination of the schedules of individual nodes.

The schedule of node j at time interval i is defined by the following function:

$$Active(j,i) := \begin{cases} 1 & \text{if node } j \text{ is scheduled to be active in interval } i \\ 0 & \text{otherwise} \end{cases}$$

To solve the problem of keeping the number of nodes constant the average number of nodes R_{avg} that are active in each time interval has to be known. This information can be computed by dividing the sum of the required node lifetimes by the remaining lifetime required for the whole network:

$$R_{avg} := \frac{\sum_{j=1}^{R} T_{rem}(j)}{\overline{T}_{rem}}$$

A solution that provides constant application quality always has a fixed number of nodes active. To minimize variations, this number should be as close as possible to R_{avg}. The deviation from R_{avg} can be computed in the following way:

$$\Delta_{active} := \sum_{i=1}^{\overline{T}_{rem}} \left| \sum_{j=1}^{R} Active(j,i) - R_{avg} \right|$$

For example, Fig. 6.10 shows all combinations of schedules for two nodes. The network lifetime is assumed to be three time intervals. Node 1 can be active for two intervals whereas Node 2 can be active for just one time interval. In this example R_{avg} is 1, i.e., a solution is optimal if exactly 1 node is active in every time interval. Therefore, there are three optimal solutions (solutions 3, 5, and 7).

Finding an optimal solution can be implemented as computing the permutations for each node and calculating the deviation from R_{avg} for each time interval. Then an optimal solution has the minimum sum of the deviations. However, this approach is inefficient because there can be a large number of schedules. Therefore, in Section 6.3.2 we present a more efficient distributed heuristic that can be run on the sensor nodes.

Balancing Energy Level Assignments

When energy level assignments are coordinated, each node continues to solve the optimization problem of Section 6.2.4. Therefore, it still optimizes its average utility over its lifetime. The only difference is that the point of time when to select each level assignment is determined by taking into account the assignments of other nodes.

Just like computing the schedule of active nodes, balancing energy level assignments can be expressed as computing the permutation of a string of level numbers. Here, however, the value of a character is only defined if the node is scheduled to be active in the corresponding time interval. In this case the value corresponds to the number of an energy level.

Again the goal is to minimize the deviation from the average. Now, however, the average utility of the energy levels has to be considered. Therefore, we define the function $Utility(l)$ that maps energy level numbers to their corresponding utility values.

The energy level of node j at time interval i is defined by the following function:

$$Level(j,i) := \begin{cases} \text{level assigned at time } i & \text{if } Active(j,i) = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Using these two functions and the function $T_{rem}(j)$, which returns the remaining required lifetime of node j, the average utility of the nodes in all time intervals can be computed:

$$U_{avg} := \frac{\sum_{j=1}^{R} \left(\sum_{i=1}^{T_{rem}} Active(j,i) \cdot Utility(Level(j,i)) \right)}{\sum_{j=1}^{R} T_{rem}(j)}$$

3a:	Node 1: -12	**5a:**	Node 1: 1-2	**7a:**	Node 1: 12-
	Node 2: 1--		Node 2: -1-		Node 2: --1
3b:	Node 1: -21	**5b:**	Node 1: 2-1	**7b:**	Node 1: 21-
	Node 2: 1--		Node 2: -1-		Node 2: --1

Figure 6.11.: Combinations of energy level schedules for two nodes

The nominator of this fraction sums up all utility values from the energy levels assigned to the nodes. The denominator, in contrast, computes how long all nodes are active throughout the network lifetime.

Again, the deviation from the average is to be minimized in order to provide constant application quality. This deviation can be computed by subtracting the overall average utility U_{avg} from the average utility of the active nodes in each time interval:

$$\Delta_{levels} := \sum_{i=1}^{\overline{T}_{rem}} |\delta(i)|$$

$$\text{with } \delta(i) := \begin{cases} \frac{\sum_{j=1}^{R} Active(j,i) \cdot Utility(Level(j,i))}{\sum_{j=1}^{R} Active(j,i)} - U_{avg} & \text{if } \sum_{j=1}^{R} Active(j,i) \neq 0 \\ U_{avg} & \text{otherwise} \end{cases}$$

We try to find a permutation of level assignments with the minimum value for Δ_{levels}. Only if a node is active, it can be assigned an energy level. In addition, to give preference to the first problem the schedule of when to activate a node has to be optimal. However, such an optimal schedule is not necessarily unique. Therefore, for the global optimum solution it might be necessary to change the activation schedule if this could improve Δ_{levels}. A different (optimal) activation schedule might lead to different energy level assignments since other nodes can be active at the same time.

Fig. 6.11 continues the example from Fig. 6.10. It assumes that Node 1 can switch for one time interval to Level l_1 and for another one to Level l_2. Node 2, in contrast can only select Level l_1. If the utility values are equal to the level numbers, the average utility U_{avg} is in this example $\frac{4}{3}$. This value cannot be achieved exactly because discrete energy levels for fixed time intervals are to be assigned. The best solutions have a deviation from the average level of $\frac{1}{3} + \frac{1}{3} + \frac{2}{3} = \frac{4}{3}$, where the $\frac{1}{3}$ terms refer to the time intervals in Level l_1 and the $\frac{2}{3}$ term to the time interval in Level l_2. As the figure shows, all three optimal assignments from Fig. 6.10 lead to the same deviation. In fact, even creating a different permutation over level assignments (the variations a and b of each solution) does not change the deviation in this simple example.

6.3.2. Realization on Sensor Nodes

This subsection describes how the problems of Section 6.3.1 can be solved on the sensor nodes. First, it gives some details about the complexity of the problem. Second, it shows how the solution can be approximated in a completely distributed way that requires the nodes only to send minimal amounts of data. Finally, it presents a greedy approach that in combination with the distributed algorithm efficiently finds a solution.

Number of Permutations

The straight-forward way to solve the problems introduced in Section 6.3.1 is to use a backtracking approach. Such an approach computes all permutations and selects one with the minimal deviation from the average. In practice, two backtracking runs are needed: one to find an optimal schedule for activating nodes and another one to find an optimal level assignment for such schedules. However, the overhead for computing such optimal solutions is significant. In addition, to ensure a network-wide optimal assignment, this computation would have to be done centrally. Therefore, not only the computational overhead would be significant but also the amount of data that is transmitted.

Even if just one of the two problems is solved, computing the optimal deviation for all permutations requires a large number of such computations. For example, Fig. 6.10 shows that even for a very simple problem there is already a considerable number of schedules possible. In general, for each node j there are S_j permutations of the activation schedule:

$$S_j := \frac{\overline{T}_{rem}!}{T_{rem}(j)! \cdot \left(\overline{T}_{rem} - T_{rem}(j)\right)!}$$

This equation computes the number of permutations for two distinct kinds of elements in a multiset. One kind of elements corresponds to the active time intervals of node j with a multiplicity of $T_{rem}(j)$ whereas the other one represents the inactive time intervals with a multiplicity of $\overline{T}_{rem} - T_{rem}(j)$.

If there are R nodes to consider, there are $\overline{S}$ combinations of their schedules:

$$\overline{S} := \prod_{j=1}^{R} S_j$$

If a backtracking approach is used, both the factorial and the product lead to a fast growth of the number of possible combinations. For example, for a network of just five nodes where every node can be active for five time intervals and the overall network lifetime is 10 time intervals, $\overline{S}$ has a value of 10^{12}. If the network should be active for 20 time intervals and all other numbers are not changed, more than 10^{20} permutations have to be considered. Furthermore, an additional large number of permutations would have to be computed for balancing energy level assignments as well.

Obviously, such a problem cannot be solved on a resource-constrained sensor node. In fact, checking all permutation seems to be virtually impossible for realistic problem sizes. Therefore, even in a hybrid network with some more powerful devices or with some nodes connected to a PC, a backtracking approach cannot be used to find the optimal solution. We address this problem by introducing a distributed heuristic that can be applied efficiently even on sensor nodes. Although its results might not be optimal, they are in general good enough to significantly reduce the variations in application quality.

Distribution in the Network

Basically, to implement an assignment algorithm for the sensor network there are two alternatives that do not require a global view of the network: having a cluster head assign schedules to its neighbors and computing a local schedule on each node while using the neighbors' schedules as constraints.

Although the first solution is promising, it has the disadvantage that some nodes – i.e., the cluster heads – have to do significantly more computation and communication than the others. Therefore, their energy budget is burdened most. This problem could be alleviated only by switching the role of a cluster head at runtime, which would probably require to change the boundaries of clusters as well. This would add significant overhead to the system. Furthermore, computing an exact solution to the problem for all nodes of the cluster would probably increase complexity beyond the capabilities of the sensor nodes. Because of that they could only approximate an optimal solution. Nevertheless, we have explored this approach in a diploma thesis [Ostermann 2007].

However, we finally selected the second alternative, where each node computes its own schedule. Although this approach might not find a globally optimal solution, either, it schedules the current node optimally with respect to the schedules of neighboring nodes. In addition, it has the advantage that its overhead is comparatively small: each node only has to compute its own schedule and broadcast it to its neighbors. Depending on the number of energy levels, it is sufficient to store and transmit a few bits for each time interval. Moreover, to avoid too frequent level changes and to further reduce the overhead, longer intervals than those in Section 6.2.4 can be used – at the cost of reduced agility. If these longer intervals are used, a node can send its schedule in just a single message.

Fig. 6.12 shows an overview of our distributed algorithm. It is repeated for each such longer time interval. First, the nodes wait for messages from their neighbors. Before

```
While $\overline{T}_{rem} > 0$
  Listen for schedules from neighbors and wait
    On receive:
      Set schedule of neighbor to values received
  Compute local schedule
  Broadcast local schedule to neighbors
  Wait until next computation round
```

Figure 6.12.: Algorithm for the distributed optimization

the receivers of such a message do their own computation, they set the schedules of the sender to the values received. Nodes from which no schedule has been received so far are completely ignored and are not included in the optimization. Therefore, the only schedule that can be modified is the one of the local node itself. This reduces the complexity of the computation greatly. In addition, this approach considers the specific neighborhood of each node even if neighborhoods are not completely identical. However, since only the schedule of a single node can be modified, the solution is probably not globally optimal. Finally, the node sends its schedule to its neighbors and waits for the next computation round.

Even though this optimization is done for just a single node, it is too expensive for resource-constrained sensor nodes to compute all possible permutations. For example, computing all valid solutions for 20 time intervals and seven neighboring nodes can take several minutes even if run on a PC. Therefore, we pursue a different approach that greedily activates nodes in time intervals such that the deviation is minimized.

Each node repeats its computation periodically in order to deal with changes in the local optimization results and with failures of other nodes. In later rounds previous results from all neighbors can be used to further improve the assignment even if a node has not updated its schedule for the current round yet.

Just like the application-specific coordination introduced in Section 6.2.4, this optimization is not fixed to particular levels: if during a later local optimization an energy level is no longer part of the result and if the distributed assignment has not been updated yet, a node might select another energy level from its local optimization instead. Therefore, there can be some variations in energy level assignments that are not detected by the distributed computation.

Fig. 6.13 shows an example of how the distributed computation is done. The example assumes that all five nodes can communicate with each other and that the optimization is done in the order of the node IDs. Furthermore, the network lifetime is assumed to be five time intervals and each node can be active for three of them (two in Level l_2 and one in Level l_0).

Since Node 1 just has to consider its own schedule for the first computation round, any assignment is as good as any other one. After the computation it sends its schedule

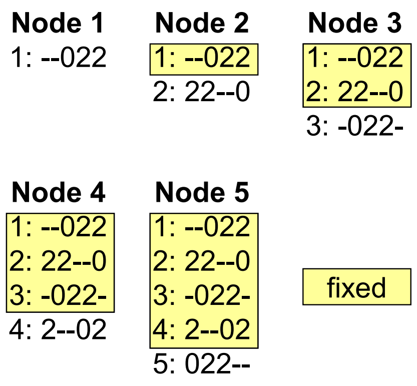


Figure 6.13.: Distributed computation of energy levels

to the other nodes. Node 2 uses this schedule when computing its own activation times and energy level assignments. Of course, it does not modify the schedule of Node 1. Then it locally broadcasts its own schedule to the other nodes. Likewise, the remaining nodes compute their schedules using those of the first ones as input.

In this example the results for both the activation schedule and the level assignment are optimal. In each time interval three nodes are active, which is also the average R_{avg} that the algorithm has tried to achieve. Likewise, the average level is – in this example – equal to the optimum U_{avg} of $\frac{4}{3}$.

Greedy Approach

To compute an optimal schedule of a single node, it is not necessary to check all permutations of its schedule. In fact, it is sufficient if a greedy approach is used that selects those time intervals (one after the other) where the deviation from the average is the smallest one. Fig. 6.14 shows this algorithm that computes the activation schedule of a node.

The algorithm first calculates the average R_{avg} as described in Section 6.3.1. This computation already assumes that the node is turned on for its required lifetime even though it has not been assigned time intervals yet. Thus R_{avg} corresponds to the actual target value.

The algorithm initializes its variables to deactivate itself in all time intervals (all values set to 0). Then it computes the deviation values from R_{avg} if the node is active in just one interval. This computation is done for all time intervals and the node stores the values with the minimum deviations in Min_Deltas. These numbers refer to the intervals when the node should be active. Finally, it schedules itself to be active in the intervals corresponding to these values.

For example, if in Fig. 6.13 Node 5 tries to assign its active time slots, it first computes the overall average of active nodes per time interval, which – in this example – is 3.

```
// initialization
Compute $R_{avg}$
For $\tau = 1$ to $\overline{T}_{rem}$
   Set activation schedule in interval $\tau$ to 0
$Min\_Deltas = \{\}$

// determine intervals with minimum deviations
For $\tau = 1$ to $\overline{T}_{rem}$
   Set activation schedule in interval $\tau$ to 1
   Compute $\Delta_{active}$
   If $|Min\_Deltas| < T_{rem}$
      Add ($\tau$, $\Delta_{active}$) to $Min\_Deltas$
   Else if $\Delta_{active}$ smaller than largest value in $Min\_Deltas$
      Remove largest element from $Min\_Deltas$
      Add ($\tau$, $\Delta_{active}$) to $Min\_Deltas$
   Set activation schedule in interval $\tau$ to 0

// activate node in these intervals
For all $(\tau, \Delta_{active}) \in Min\_Deltas$
   Set activation schedule in interval $\tau$ to 1
```

Figure 6.14.: Algorithm to compute the local schedule

It then assumes that it is active in one time slot and computes the deviations from the average. Regarding the other nodes, two of them are active in the first three time intervals whereas already three nodes are active in the last two intervals. If Node 5 is activated in one of the first three intervals, the overall deviation Δ_{active} is 2. If it is activated in one of the last two intervals, however, the deviation from R_{avg} is 4. Therefore, since the node is required to be active for three time intervals, it schedules itself to be active in the first three intervals.

This algorithm always finds an optimal solution subject to the schedules of the neighbors and considering the fixed-length time intervals: If the node selects a time interval with the smallest deviation from R_{avg}, this interval has to be part of an optimal solution since it corresponds to the summand of Δ_{active} with the minimum values. Because all the summands of Δ_{active} are non-negative, the sum itself becomes minimal if all of its summands are minimal. The only way to influence the deviation from the average is to make the node active. In addition, the node has to be scheduled for exactly its remaining lifetime. Therefore, there is no other assignment that could lead to a smaller overall deviation.

The efficiency of this algorithm is much better than that of an algorithm that computes all permutations. In fact, it just has to compute $\overline{T}_{rem}$ values for the deviation. If the overhead of a straight-forward solution to maintain the list of time intervals with minimum deviation is considered, the overall complexity of this algorithm is $\mathcal{O}(\overline{T}_{rem}{}^2)$. Compared to S_j, the number of permutations of the schedule, this is a significant improvement since S_j depends on the factorial of $\overline{T}_{rem}$.

The problem of balancing energy levels can be solved analogously because in each local optimization result the Simplex algorithm returns at most two energy levels with non-zero time durations. This directly corresponds to the problem of activating nodes with the two states "on" and "off". Here, however, only time intervals are considered in which the node is scheduled to be active. In the beginning all of these intervals are assigned the lower energy level from the local optimization result and the target value for the average utility U_{avg} is calculated. Then the node computes the deviation Δ_{levels} from U_{avg} if it assigns its higher energy level to each interval, and selects the intervals with the minimum deviations. Just like when computing the activation schedules, an interval with a smaller deviation does not exist.

Using this algorithm and the distributed computation outlined in the previous subsection, the computational overhead for each node is minimal. Therefore, as we show in the evaluation, even resource-constrained sensor nodes can optimize their own schedule with respect to their neighbors' schedules.

Although this approach always finds an optimal solution for each of the two sub-problems subject to their constraints, its overall solution might not be the global optimum for all nodes and for both sub-problems combined. First, because of the distributed computation a node cannot modify its neighbors' schedules. Therefore, the solution might not be the network-wide optimum. Second, because assigning the active time intervals of nodes and balancing the energy levels are solved independently from each other, a node might not be active in time intervals where it could achieve the average utility more closely even if such an activation schedule might also be optimal. For example, in some time intervals only those nodes with low energy levels left might be active. However, experiments have shown that this is not a problem in practice.

6.4. Evaluation

This section evaluates the benefits and overhead of *Levels*. For this purpose we use both simple applications, which correspond to components found in more complex ones, and real-world applications.

Unless otherwise mentioned, we use the Avrora simulator again, which accurately emulates Mica2 sensor nodes. The battery voltage that the simulator makes available to the sensor nodes' voltage sensors has been recorded from individual batteries with the voltage sensor of a real sensor node. In contrast, *Levels* running on the sensor node uses a battery model based on the average of several such voltage traces (see Section 6.2.4). Therefore, this simulation setup corresponds to the situation of real sensor nodes.

Table 6.1.: Average lifetimes of sample applications for constant energy levels (in days)

Application	**Level l_0**	**Level l_1**	**Level l_2**
FFT	961	375	not used
Flash	961	296	not used
SendLPL	34.6	22.5	16.7
SendLPLRandom	34.6	27.1	22.2
SendRadioOff	948	692	not used
Voltage	7.69	6.65	5.93

6.4.1. Quality of Level Assignments

In this subsection we evaluate the quality of level assignments. To do this we use several metrics: First, we contrast the actual to the required lifetime. Second, we compare the average utility achieved in simulation with the optimal value possible. Finally, we validate our simulation results with experiments using real sensor nodes.

Simulation of Small Applications

The small applications that we use for this evaluation represent parts which can also be found in larger ones. FFT periodically computes a Fast Fourier Transform, Flash stores data into flash memory, and SendRadioOff turns the radio chip only on for sending (short) messages but does not listen for any messages itself. Unlike SendRadioOff, SendLPL uses low-power listening [Polastre et al. 2004] and thus sends messages with longer preambles. In addition, it includes another energy level where a second periodic message is sent. SendLPLRandom is similar to that. However, instead of periodically running this code it waits for a random time and then sends a random-length burst of messages. Finally, Voltage periodically sends messages with its current voltage reading and, on the highest energy level, toggles its LEDs every thirty seconds. We used this application also for experiments with real nodes. Because of the time constraints of our experiments we intended to create a particularly energy-intensive application here.

Except for SendLPLRandom all of these applications execute some tasks periodically. This code has been encapsulated in an energy level which can be deactivated if necessary. In this case, however, the applications will no longer perform their actual tasks. SendLPLRandom, in contrast, represents an event-based application. To simulate events, it waits for a random time (up to one hour) before running its optional code.

In most applications the utility values have been set to 0 for Level l_0, 1 for Level l_1, and 2 for Level l_2. For SendLPL and SendLPLRandom, however, the utility of level l_1 and l_2 has been increased slightly to 2 and 3, respectively. Because of the greater gap to l_0 and the small difference to l_2 these two applications preferably switch to l_1 if they cannot stay in l_2 all the time.

Table 6.1 shows the simulated lifetime of the applications when a constant energy level has been set. We validated some of the shorter lifetimes with real sensor nodes. The results of these experiments differed by at most 2.2 % from the simulated values. We attribute these differences mostly to variations in the capacity of the batteries used and slight deviations in the energy model of the simulator.

The table shows that our evaluation includes very low-power applications with a maximum lifetime of several years like FFT, Flash, and SendRadioOff as well as applications like SendLPL, SendLPLRandom, and Voltage which have a high energy consumption. In addition, the lifetime of the nodes varies significantly for different energy levels. Depending on the application, the lifetime can be extended by between 30 % and 225 % if only the lowest level is active.

Levels will not be able to meet a lifetime goal if the lifetime requested cannot be possibly achieved when the application does not already start in the lowest level. For example, for one of our simulated batteries, SendLPL has a maximum lifetime of 50,318 minutes in level l_0. If a lifetime of 50,000 minutes is requested for this application and if the initial energy level is l_2, *Levels* switches to level l_0 as soon as possible. Nevertheless, it can only achieve a total lifetime of 49,348 minutes and, therefore, fails to meet the requested lifetime goal. However, cases like that are somewhat artificial since using *Levels* does not make much sense if the lifetime goal can hardly be met in the lowest level. Therefore, in this evaluation we focus on more realistic lifetime goals where better application quality is actually possible.

We have set such goals for all sample applications and selected the lifetimes such that the first run could be completed in the maximum level with most simulated batteries. Fig. 6.15 compares the average lifetime when simulating different batteries (including 95 % confidence intervals) with the lifetime requested. In each of these simulations *Levels* was able to meet the lifetime goal. In addition, the size of the confidence intervals is in almost all cases smaller than 4 % of the total lifetime. This is less than the confidence intervals of the simulated battery capacity, whose size is about 6.4 %, since *Levels* adjusts to the actual battery discharge curve.

Furthermore, the nodes did not live much longer than requested. Therefore, the applications were able to provide almost optimal quality subject to the constraints present. The largest differences relative to the lifetime achieved can be observed most often for the simulations with the smallest lifetimes because they have been chosen in a way that the applications can stay in their highest levels for the complete lifetime. Therefore, the node cannot possibly consume all the energy available. For example, in the first simulation run, SendLPL lives about 20 % longer than requested although *Levels* stays in Level l_2 for almost the complete simulation.

Besides that, the relative differences between the required and the actual lifetime are the largest ones for SendRadioOff (5.9 % on average). Due to the inaccuracies in the estimation of the battery's remaining energy, *Levels* defers the computation if only a very small amount of energy has been consumed in optional energy levels. Therefore,

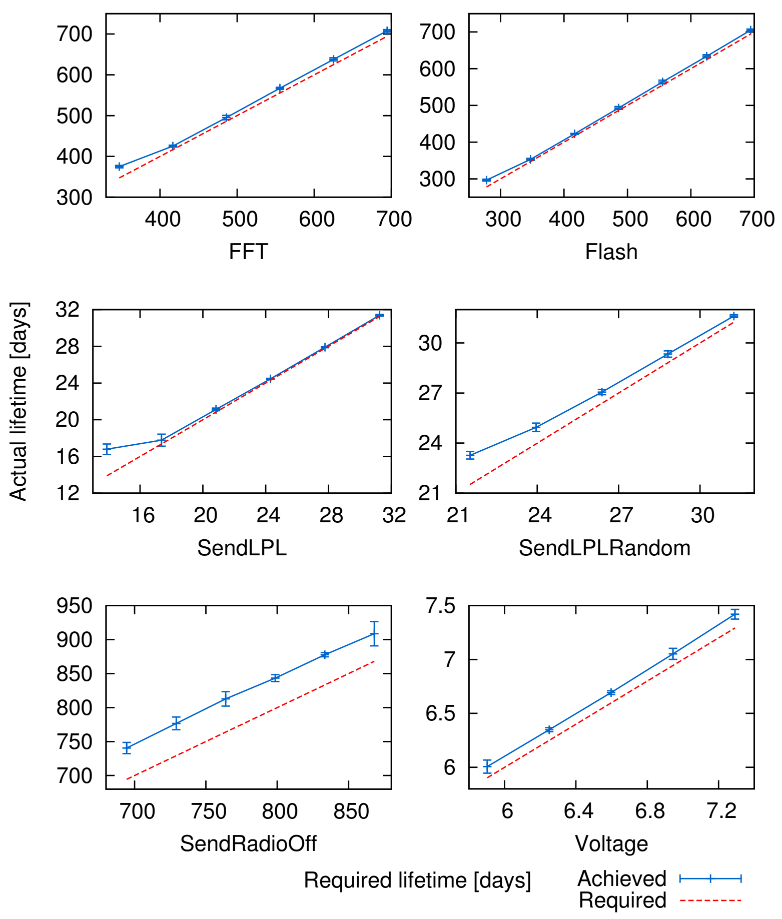


Figure 6.15.: Required lifetime vs. lifetime achieved (including 95 % confidence intervals)

Levels can only execute a small number of level computations and cannot switch long enough back to a higher level when the energy reserved as a safety overhead becomes available near the end of the required lifetime. Since this application consumes in Level l_1 just 0.14 mW more than in level l_0, it can take more than 45 days until a new level assignment is computed, which reduces the agility of level adjustments. This delay could only be reduced significantly with more accurate hardware to measure battery capacity.

SendRadioOff is an extreme example for the amount of energy consumed where meaningful computations are almost impossible. As other long-lived applications show, a small increase in power consumption is enough to obtain considerably better results. For example, FFT, for which the actual lifetime is much closer to the required one, consumes in Level l_1 just 0.60 mW more than in Level l_0.

This problem of SendRadioOff is also shown in Fig. 6.16. This figure compares for a single battery the average utility value achieved within the requested lifetime to the optimal average. This optimum has been computed offline by solving the same

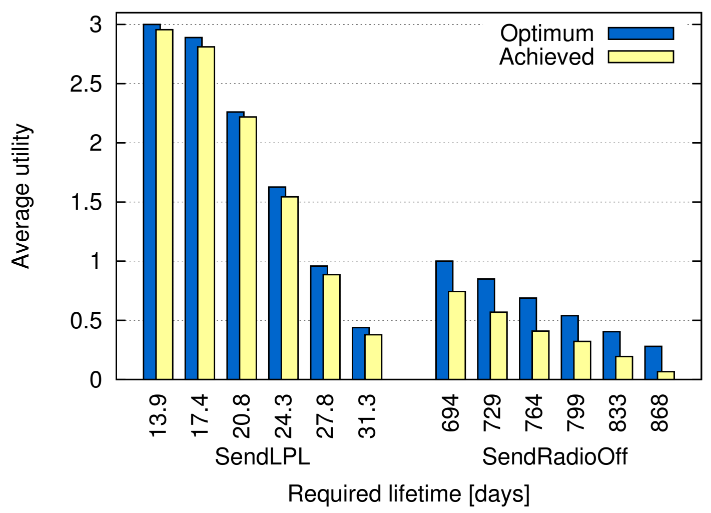


Figure 6.16.: Average utility throughout the required lifetime

linear programming problem as *Levels*. However, it has been calculated only once using exact information about the battery capacity and the energy consumption of the different levels.

The figure shows that SendLPL achieves the optimal utility value almost perfectly despite the inaccuracies present on the sensor nodes. For SendRadioOff the difference is greater (about 0.24 utility units) because of the small number of level computations described above.

Experiments with Mica2 Nodes

We validated the simulation results of the Voltage application in experiments with real hardware using Mica2 sensor nodes. To prevent side-effects from slight variations in energy consumption we calibrated the energy model used for profiling with a multimeter to the specific sensor nodes used and created a separate battery model for each node.

Fig. 6.17 shows the actual lifetime achieved by the motes when varying the required lifetime. We define as the lifetime the time a neighboring node was able to receive periodically transmitted packets, which were sent irrespective of the current energy level. In most experiments the nodes met their lifetime goal. However, in the last experiments we ran – some of those with a lifetime of 10,000 minutes (6.94 days) – most nodes failed early although in previous experiments they accurately achieved this lifetime. For this time value these failures reduce the average lifetime and increase the size of the confidence interval. We had purchased the batteries used in these (failed) experiments several months after the ones used to build the battery model. A detailed analysis of the recorded voltage readings showed that the nodes expected to have significantly more energy left than actually available. We attribute this to differences

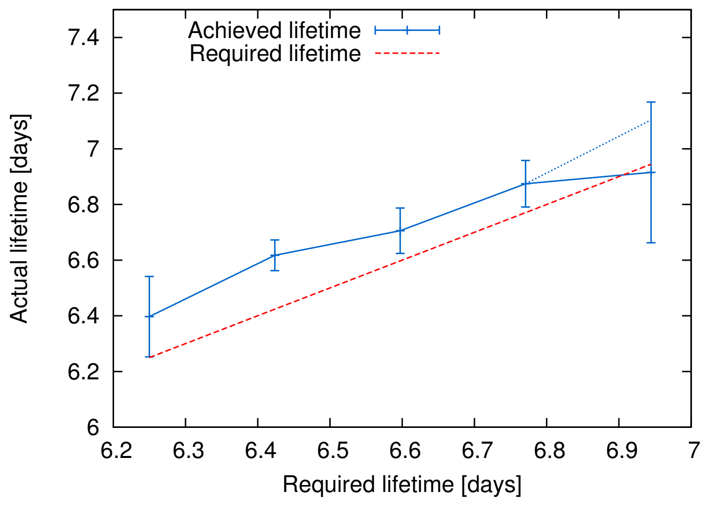


Figure 6.17.: Lifetime of the experiments with Mica2 nodes (including 95 % confidence intervals)

in the properties of the batteries. In fact, after updating our battery model a lifetime of 10,0000 minutes could be achieved again. This shows that a good representation of the battery characteristics is needed for *Levels* to accurately meet lifetime goals. If the experiments that failed are not considered for this lifetime, the lifetime achieved would be on average 10,229 minutes (7.10 days). This is shown with the dotted line segment.

The lifetime achieved by the sensor nodes in all other experiments was between 1.1 % and 6.5 % longer than the lifetime requested. Considering variations in battery capacity and the relatively small number of possibilities to adjust the level for this short-lived application these numbers are excellent. Although the variations due to external influences are greater here, the results correspond to our simulations. Therefore, they validate that the models used in the simulator capture the relevant factors of real deployments.

Using our original approach, the sensor nodes start ideally in their highest energy level and only switch to lower levels later if the lifetime goal can no longer be met. Thus they minimize the number of level changes and avoid frequent changes in application quality. Due to inaccuracies on the sensor nodes, this best case cannot be achieved at runtime.

This is shown in Fig. 6.18 that visualizes the energy levels assigned in one of our experiments with a Mica2 sensor node. As expected, the node starts with the highest level (level l_2) and after two days switches to the lowest level (level l_0). Then, however, near its target lifetime it switches back to higher levels. This behavior is due to the safety factor in our computations: The node detects that its estimation has been too conservative and tries to consume the energy available to provide the best application quality possible.

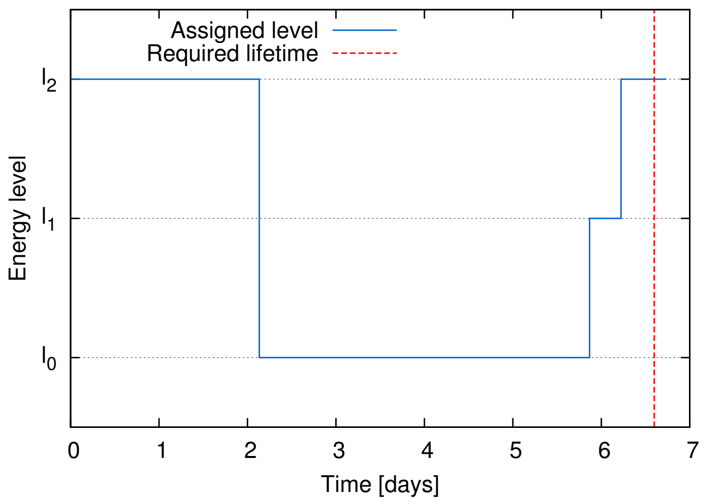


Figure 6.18.: Level assignment over time

Although this behavior differs from the ideal case, the number of level changes is still very small; in this experiment just three such changes occurred within several days. Depending on the energy consumption of the application and the requested lifetime more changes can be necessary. Nevertheless, 70 % of all the simulations presented in the previous subsection required 10 or less actual level adjustments.

6.4.2. Real-World Applications

In this subsection we show how *Levels* can be applied to real-world applications. For this purpose we selected monitoring of volcanoes [Werner-Allen et al. 2006b]. In this application there is usually no redundancy in the network topology and the required duration of the experiment is known in advance. Replacing batteries is extremely difficult due to the inaccessible deployment location. Moreover, large parts of each node's energy are used to power the sensor interface board. Therefore, if a node stops sampling data itself but continues forwarding data from other nodes, its lifetime can be extended significantly and network connectivity can be preserved much longer.

As a concrete example of this class of applications we chose the system used at Reventador ("Volcano") [Werner-Allen et al. 2006a]. This system is a complex application that has been tested in real-world deployments. It stores sensor readings to flash memory and the base station can then request stored data. In addition, Volcano includes an in-network detection of volcanic eruptions.

Again we use the Avrora simulator for the evaluation. However, since this simulator did not include the custom sensor interface board used by this application, we had to add its energy consumption to the simulator's energy model. From the information available we assumed for the sensor board a current draw of 40 mA. Furthermore, Volcano has been originally created for Telos B nodes while the prototype implementation of *Levels* assumes the Mica family of sensor nodes. Therefore, we ported the application to this hardware family. However, in order to keep changes to the application small we simulated for this evaluation a fictitious Mica2-like node that is – like the Telos B nodes – equipped with more RAM.

The behavior of Volcano depends on the eruptions detected by the sensor nodes. We simulated these eruptions at random intervals such that on average one event occurred every 30 minutes. However, like in the real deployment not all of these eruptions were actually reported by the nodes if they, for instance, stopped sampling to transfer some data.

In the deployment of the application [Werner-Allen et al. 2006a] some batteries with higher capacities than those in our battery model were used. Therefore, our simulated lifetimes are significantly shorter than those reported there; this reduction in lifetime is not due to *Levels* and can also be observed when simulating the original application with the parameters of our batteries.

In its original version Volcano does not include code to achieve a user-defined lifetime goal. Therefore, we specified some optional functionality using our energy level abstraction. Since sensing is the largest single energy consumer, we put this code into a separate energy level. If it is deactivated, the nodes turn off the energy-expensive sensor interface boards and stop analyzing, storing, and transmitting their data. However, they still fully participate in routing and thus forward data from other nodes.

We defined energy levels in two nesC modules that were then mapped to a single level in the application. Only minor changes to the existing code were necessary: about 20 lines of code had to be added or modified. Some larger effort was, however, needed to write the profiling functions since no suitable unit test drivers were available. The size of this module is less than 200 lines of code. In addition, we were able to copy almost the complete nesC wiring from the actual application and reuse it for energy profiling.

Fig. 6.19 shows the average lifetime achieved by this application. In total we simulated 150 sensor nodes and none of them failed before its lifetime goal. However, since in this complex application the behavior of the nodes depends on network packets received and random events detected, accurately predicting future energy consumption from past data becomes more difficult than with the simple applications of Section 6.4.1. Therefore, the variations can be greater for this application. This is shown with the confidence intervals in Fig. 6.19, whose sizes are between 3.6 % and 6.0 % of the lifetimes requested. In addition, *Levels* is not able to completely consume the energy kept as a safety buffer and the nodes live on average 12.4 % longer than required. This number could be reduced, however, by adjusting some parameters of *Levels*. Nevertheless, considering the lesser predictability of this application even these results are encouraging.

Just like for Volcano, *Levels* could also be applied to other existing applications. To get a better understanding of the effort needed for this change we analyzed the

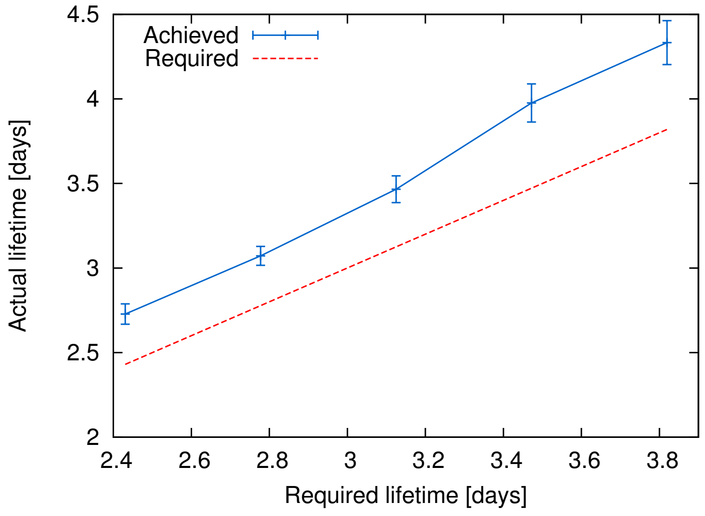


Figure 6.19.: Average lifetime for Volcano (including 95 % confidence intervals)

structural health monitoring application used at the Golden Gate Bridge [Kim et al. 2007]. For this application the sensor board is also one of the main energy consumers. Therefore, by applying energy levels just like in Volcano the lifetime of this application could be significantly extended. Similar effort as for Volcano would be needed for this modification, i.e., changing about 20 code lines in the application and writing measurement functions of less than 200 lines.

6.4.3. Distributed Assignment

This section evaluates the behavior of our distributed coordination algorithm. The most important metric for this algorithm is the deviation from the average since minimizing this value is the overall optimization goal. Therefore, we run different algorithms and compare this value both for the number of active nodes and for the average energy level assigned.

First, we use the original approach from Section 6.2.4 that does not balance energy level assignments among nodes: It always starts with the highest level first and switches to lower levels only later. In addition, for this evaluation we assume that it turns on all nodes for their required lifetime immediately after deploying the network.

Second, we use the randomization scheme from Section 6.2.4, which randomly decides which level from the local optimization result should be selected first. This approach does not require any additional optimization or coordination.

Third, the distributed greedy approach from Section 6.3.2 computes a solution that takes into account the schedules of the neighbors.

Fourth, as a first benchmark, we compute a solution using a backtracking algorithm in combination with our distributed assignment algorithm. Each node only can modify its own schedule; the schedules of its neighbors are fixed. Therefore, the solution found might not be the global optimum for all nodes. Since this computation is done for just a single node in each case, it is actually possible to compute all permutations of its schedule on a more powerful device like a PC. In addition, this approach adjusts the activation schedule if an equivalent solution exists that allows for a better level assignment.

Finally, as a second benchmark, we have approximated a solution that is optimal for the complete network using simulated annealing [Kirkpatrick et al. 1983]. Simulated annealing has been developed as a meta-heuristic for optimization problems. It has been inspired by annealing in metallurgy, where some material is temporarily heated again in its cooling process in order to increase the size of its crystals. Basically, simulated annealing follows a greedy approach that modifies the current solution and selects the new solution only if it is better than the previous one. However, transferring the annealing concept, a solution is selected with some probability even if it is not better than the previous one. Therefore, simulated annealing avoids being stuck in local optima. Using a heuristic is necessary since the complexity for realistic problem sizes is too high to compute the exact optimum. However, for small problems with the activation schedule we were able to verify that the solution found by this algorithm is the actual optimum.

All previous evaluations in this chapter were done with just a single or few sensor nodes. Thus they could be easily simulated with Avrora. However, if a greater number of nodes is run for a long lifetime, the performance of Avrora is not sufficient. Therefore, for the simulation of distributed assignments, which need more nodes to get meaningful results, we used a custom simulator that does not completely model the actual Mica2 hardware. Instead, it just provides the functionality to solve the problems of Section 6.3. Furthermore, it does not recompute a solution to the local optimization problem but uses a precomputed energy level assignment. Therefore, it cannot reflect the behavior when *Levels* switches back to a higher level near the end of its lifetime, for example. However, for comparing different algorithms that all operate on the same local optimization results, such functionality does not seem to be needed.

For these experiments we set the network lifetime to 20 time intervals. In a real application each of these intervals may include several local optimization rounds that try to maximize the average utility of the node. The required lifetime of each node was set to 8 time intervals. Half of the nodes stayed for 4 time intervals in an energy level with a utility of 2 whereas the other half stayed in this level for just 2 intervals. For the rest of their lifetime all nodes switched to a level with a utility of 0. In our experiments we placed the nodes randomly in a fixed area and varied the total number of nodes. This way we ran simulations for different network densities.

In Fig. 6.20 we show Δ_{active}, the deviation of the number of active nodes from the average. The deviation of our original approach that activates all nodes first is – as expected – the largest one since all nodes are scheduled to be active in the same time

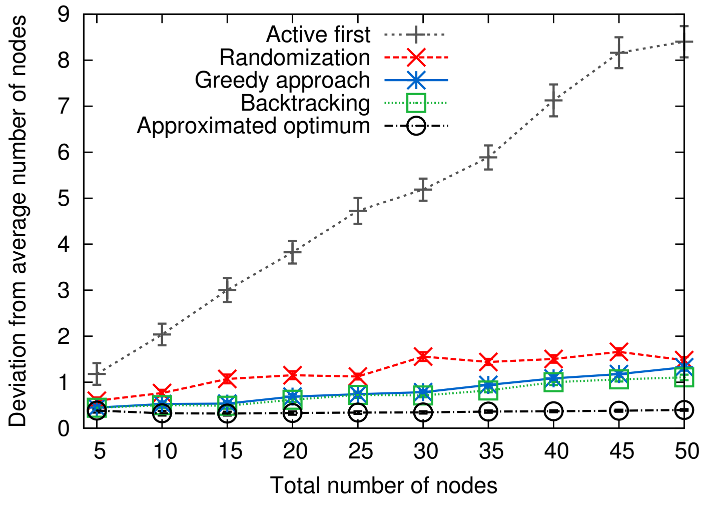


Figure 6.20.: Average deviation of the number of active nodes from R_{avg} (including 95 % confidence intervals)

intervals. The deviations increases with greater node densities since with more nodes in the neighborhood the absolute difference of the number of active nodes also grows.

For the randomization approach the results are significantly better. In addition, if the number of nodes is increased, the average deviation only increases slightly. The reason for this reduced increase is that with more nodes it becomes more likely that the random schedules are assigned in a way that in each time interval approximately the average number of nodes is active. Therefore, the actual number of active nodes is close to the average.

Although the results of the randomization scheme are already better than those of the original algorithm, they can still be improved if some coordination among nodes is possible. These approaches consider the specific neighborhood – which is also used for computing the average deviation in the figure – instead of the complete network. As the figure shows, the results of the backtracking algorithm and of the greedy approach are almost equal. This is as expected because they both compute a solution subject to the same constraints. Small differences can be attributed to suboptimal local assignments since each node can only adjust its own schedule.

Like the random approach, the results for these two algorithms increase only slightly with higher node densities. This is because with more nodes a better coordination can be performed.

For the simulated annealing approach, the deviation does not increase with higher node densities since the globally optimal solution is approximated. For the other algorithms, however, the schedules of the neighbors are fixed constraints, and only the schedule of the local node can be modified. If the node density increases, the weight of a single node decreases and more nodes are needed to balance the assignments and

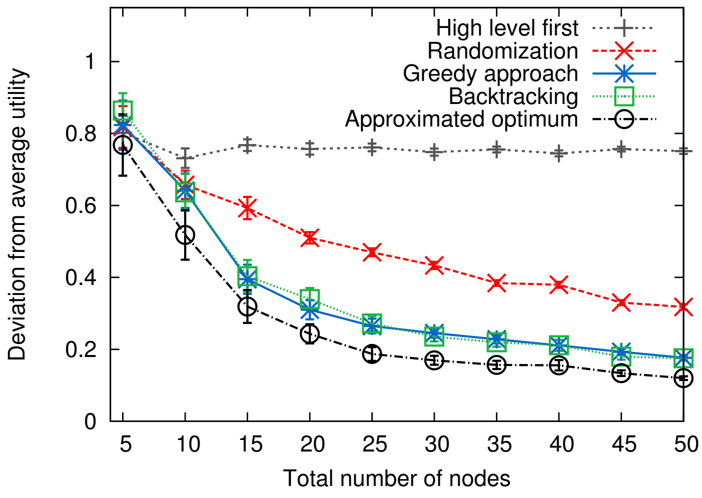


Figure 6.21.: Average deviation of the network-wide utility from U_{avg} (including 95 % confidence intervals)

get closer to the average. Since the neighborhoods of the nodes are not identical, the deviation increases slightly for both the greedy and the backtracking approach.

Fig. 6.21 shows the results for balancing energy levels. Here the values do not increase with higher node densities because this figure shows the deviation from the average utility, which is divided by the number of active nodes, instead of the absolute difference in the number of active nodes. Because the time spent in the energy levels varies and because of the nodes' placement in the topology there are some small variations. This is directly reflected by the results of our original approach, where all nodes start in their highest energy level. Since with this approach nodes do not take into account the energy level assignment of their neighbors, the deviation from the average utility is almost constant.

In contrast, the deviation of the other approaches decreases if more nodes are present. The reason for this is that with more neighbors these approaches are able to approximate the average more closely. As the figure shows, the coordinating approaches are in almost all cases better than the randomization scheme. Only in very sparse topologies with less than two concurrently active nodes in the neighborhood (simulations for 5 and 10 nodes in total) the random approach might be slightly better. Since the coordination approaches operate only in the local neighborhood, they cannot balance the assignments for such a small number of neighbors. The randomization approach, in contrast, profits from the fact that all nodes throughout the network assign their schedules randomly.

Again the greedy approach is virtually equivalent to the backtracking algorithm, as their overlapping confidence intervals show. This is notable since the latter algorithm solves – unlike the former one – both sub-problems together and adjusts the activation schedule if this reduces the deviation for the utility of energy levels.

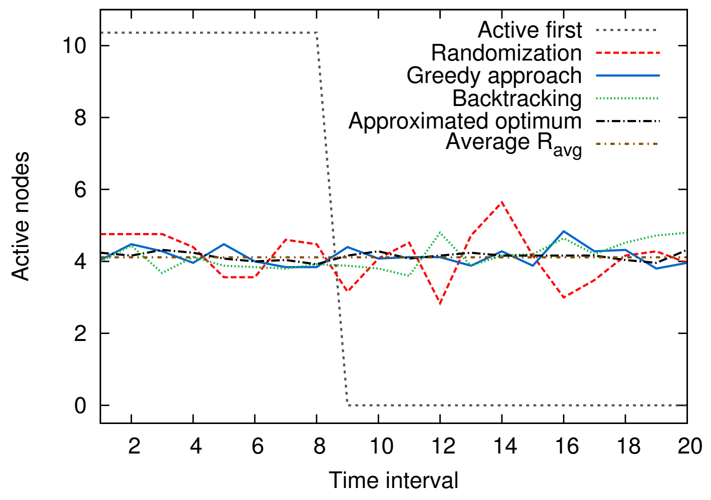


Figure 6.22.: Active nodes over time for a sample run of 25 nodes

Here, the solution of simulated annealing is again better than the other approaches. This is not surprising since this approach approximates the network-wide optimum rather than solving the problem just with local knowledge on each node. Nevertheless, as the results show, the distributed approaches are very close to the optimum. However, it should be mentioned that the approximated optimum just considers a single solution for the activation schedules. Therefore, slightly better results might be possible here if the assignments of both the activation schedules and the energy levels are solved together.

Fig. 6.22 shows how many nodes are active in each time interval for a sample run of 25 nodes. Naturally, the approach that activates all nodes first differs most from R_{avg} whereas all other approaches oscillate around this average. These oscillations are slightly greater for the randomization scheme, which leads to the greater overall deviation. The approximated optimum with simulated annealing, however, shows – as expected – the smallest deviation.

If a node is turned on and off or if its energy level changes, it probably needs some time to adjust to its new task. Therefore, the number of such changes should be kept small. In our basic approach of Section 6.2.4 this is done by starting in the highest level and by only switching to a lower level if the lifetime goal cannot be met otherwise. For our distributed and randomized approaches, however, this number will increase because level changes are used to balance assignments in the network. In addition, it has not been one of the goals of the optimization to minimize these changes.

Since in these simulations the level assignments have been set to fixed values, the original algorithm changes the activation state and the energy level just once after starting in the highest level. The other approaches require slightly more changes. For example, on average the greedy approach activates each node 3.2 times in our simulations and switches from one energy level to another one 3.0 times.

Table 6.2.: Runtime overhead of local optimizations

Optional code block	91 μs
Check-only for optional code block	11 μs
Adding continuous energy (2 levels)	30 μs
Adding continuous energy (5 levels)	107 μs
Adding continuous energy (10 levels)	235 μs

Nevertheless, the number of changes is comparatively small in the distributed approach since we use longer time intervals here than those for the local optimization. Within each such long time interval there are usually no changes unless the local optimization result from the Simplex algorithm has changed significantly.

6.4.4. Runtime Overhead

This section evaluates the runtime overhead of *Levels*. First, we show that the overhead for the purely local optimization, where there is no coordination among nodes, is almost negligible. Second, we demonstrate that adding distributed coordination does not add much overhead.

Local Optimization

Since with the purely local version of *Levels* each node determines its energy level independently from other ones, it does not have to send any radio messages. This helps to make *Levels* usable with low-power applications. Therefore, the only increase in energy consumption can be attributed to computational overhead. There are three sources for this overhead. First, whenever a code block belonging to an energy level is about to be executed, the system has to check if the level is active and has to add the block's energy consumption to its internal variables. Second, it accumulates continuous energy consumption to overall energy consumption every few seconds. Finally, every few hours *Levels* tries to adjust the current energy level with the Simplex algorithm.

To evaluate the first kind of overhead, i.e., the overhead associated with each optional code block, we instrumented a simple application that makes use of the energy level abstraction. By simulating this application we were able to measure the CPU overhead in a controlled setting.

Table 6.2 shows the result of this experiment. The overhead associated with every code block is comparatively small: it is just 91 μs. However, sometimes the energy consumed by this overhead might still outweigh the energy consumed within the code block. In these cases the runtime system just checks if the code should be executed without adding its energy value. This takes only 11 μs. Therefore, even in this case when *Levels* is of less use, its runtime overhead still does not dominate the energy consumption of the code that it controls.

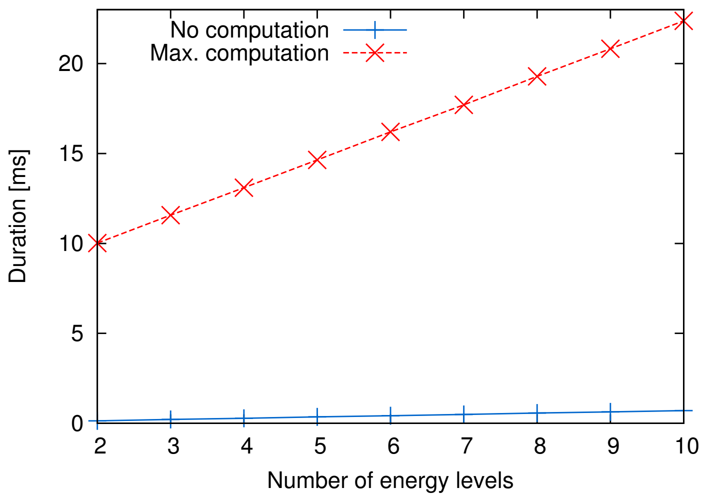


Figure 6.23.: Duration of level adjustment

Table 6.3.: Effect of runtime overhead for local optimizations on node lifetime

Application	**Lifetime with *Levels***	**Reduction**
Voltage Level l_0	7.687 days	0.0 %
Voltage Level l_1	6.648 days	0.0 %
Voltage Level l_2	5.932 days	0.0 %
FFT Level l_0	944.4 days	1.8 %
FFT Level l_1	372.7 days	0.7 %

In another experiment we measured the overhead when accumulating continuous energy consumption. Here the results depend on the number of energy levels defined in the application. Although individual software components might all define their own energy levels, we expect that application developers will combine them when creating the overall application. Therefore, most applications will probably have less than five energy levels. As the numbers in Table 6.2 show, even for applications with twice this number the overhead is just a few hundred microseconds.

Finally, computing a new level assignment incurs the largest overhead. However, since this computation is only executed every few hours, the overhead is less critical. Fig. 6.23 shows the CPU overhead for two cases. In the first one, the energy consumed is too small compared to the current accuracy given by the battery model. Therefore, the actual computation is not performed. The other one, in contrast, shows the overhead when the computation is done for the maximum number of iterations. With more energy levels the overhead increases because for each level additional variables have to be considered both when no result can be computed and for solving the linear programming problem. Although an overhead of some milliseconds might seem significant, this computation is only executed every few hours or even days (depending on the energy consumption). Therefore, it should be acceptable for most applications.

To find out the actual effects of the computational overhead on node lifetime we

Table 6.4.: Energy overhead of distributed level assignments

Action	**Energy**
Send packet with LPL	70.9 mJ
Receive for five minutes with LPL	156 mJ
Greedy algorithm	0.199 mJ

simulated some of the test applications described in Section 6.4.1 with and without our runtime system doing its computations. As the results in Table 6.3 show, for short-lived applications with a lifetime of only a few days, the energy overhead of the computation does not result in a detectable decrease in node lifetime. Even for extremely low-power applications with a lifetime of several months or even years, the CPU overhead of our runtime system leads to a reduction in lifetime of less than 2 %.

Distributed Assignment

The overhead for coordinating assignments is also very small, especially since this optimization is run less frequently and requires only minimal input from the directly neighboring nodes. Table 6.4 summarizes the main components of the overhead for a typical low-power application. Again the numbers have been obtained using Avrora.

The table assumes that the radio has to be turned on for five minutes just to exchange the schedules. During this time we make use of low-power listening [Polastre et al. 2004] (with 1 % duty cycle) that increases the length of the preamble in order to reduce the time receivers have to listen for radio messages.

If 20 time intervals are used, a node's state can be transmitted in a single network packet. In applications with as many energy levels as 128 – much more than expected in real-world applications – for each time interval just a single byte is needed: 1 bit to represent the on/off state and up to 7 bits for the energy level assigned to the time interval. Sending such a message including the longer preamble for low-power listening consumes on the Mica2 platform 70.9 mJ.

Furthermore, nodes have to be able to receive network packets for coordination. In loosely synchronized networks it should be sufficient to turn on the radio for a few minutes each time when a node expects to receive new schedules from its neighbors. Turning on the radio for five minutes with low-power listening and the accompanying switch of the CPU out of the power-save mode requires 156 mJ.

Finally, the last component of the overhead is computing the local schedule using our greedy algorithm. This computation takes for 20 time intervals 9.39 ms and requires just 0.199 mJ of energy.

In summary, the energy needed for communication dominates the overhead for computation. In total one coordination cycle requires 227 mJ. If at the beginning an initialization is performed and if 20 computation cycles are executed during the lifetime of the network, the coordination consumes altogether just 4.77 J. Compared to a total battery capacity of about 32,000 J this is negligible. Even if each message was sent several times to increase reliability, an effect on node lifetime would hardly be detectable.

6.4.5. Summary

In this evaluation we have shown that *Levels* is able to accurately achieve given lifetime goals even if the application depends on external events. However, it requires detailed information about the energy consumption and the battery discharge behavior in order provide the expected results. With relatively small changes it is possible to add *Levels* to existing real-world applications.

If some coordination in the network is available, *Levels* can balance both the active time slots and energy level assignments in a way that provides almost constant application quality over time. Furthermore, the overhead both for the local optimization and for the distributed coordination is so small that it can virtually be neglected.

6.5. Related Work

In this section we give a brief overview of work related to *Levels*. Particularly, we describe systems that take into account energy considerations for adaptation, extend network lifetime by deactivating redundant nodes, map energy consumption to code blocks, and model battery behavior.

In the realm of mobile computing, Odyssey [Flinn and Satyanarayanan 1999] monitors the available energy and adapts the fidelity of applications to meet a user-defined lifetime goal. For example, a video player switches to a differently compressed source file or reduces its window size if energy becomes scarce. Odyssey does not provide a programming abstraction like our energy levels and does not leverage simulation data. Therefore, it cannot take advantage of predicting energy consumption after adaptation. Furthermore, it has been designed for less resource-constrained devices and relies on highly accurate measurement equipment, which we cannot assume on inexpensive sensor nodes.

Similarly, ECOSystem [Zeng et al. 2002] tries to achieve a target lifetime by limiting the discharge rate of the battery. It introduces the Currentcy Model to deal with the demands of competing tasks in a multitasking system. Rather than identifying optional functionality in applications, it modifies the scheduler to execute only those tasks that have not spent their energy budget for the current round yet. Unlike our approach it does not exploit information from simulation and, therefore, has to do detailed energy accounting at runtime.

In the field of sensor networks an architecture for energy management has been presented recently [Jiang et al. 2007b]. It shares many goals with our work like, for example, the ability to specify a user-defined target lifetime and to identify optional functionality. However, since this work seems to be at an early stage, the properties of concrete implementations of this architecture are still unclear.

Probably most closely related to *Levels* is Eon [Sorber et al. 2007], which has been developed concurrently to our work. It provides a language and runtime environment for energy-aware sensor network applications. However, instead of dealing with lifetime goals Eon focuses on perpetual systems that employ energy harvesting. Therefore, it pursues the goal of balancing energy consumption and production. Eon does not make use of information from simulation and, therefore, can only estimate the energy consumption of different functionalities. In addition, it does not coordinate the functionality assignments of nodes.

TinyDB [Madden et al. 2005] allows to adapt the interval between the measurements of a query in order to meet user-defined lifetime goals. Similar to our rationale, its authors argue that in environmental monitoring scientists are more concerned about meeting a lifetime goal than about the sampling rate. Since TinyDB's programming interface is based on high-level SQL-like queries, changing the sampling rate is the only way to influence network lifetime.

There is already a large body of work dealing with the coverage problem in wireless sensor networks. This work switches redundant nodes into sleep mode to maximize the time that a given area is monitored by the network [Cardei and Wu 2006; Huang et al. 2006]. Closely related are topology control mechanisms like ASCENT [Cerpa and Estrin 2002] that switch off redundant nodes but strive to preserve network connectivity. Similarly, PEAS [Ye et al. 2003], for example, controls the network density to ensure both coverage and connectivity. Likewise, duty-cycling approaches [Giusti et al. 2007] periodically turn off nodes to extend network lifetime. Unlike *Levels* all of these approaches are only targeted to dense networks where redundant nodes can be temporarily deactivated. In addition, although they keep the application quality roughly constant, they do not have any given lifetime goals but try to maximize the time for which they provide coverage or connectivity, respectively. Finally, they do not provide more states with differing functionality – like our energy levels – and usually turn nodes just on or off.

Many network protocols already include mechanisms to reduce energy consumption. For example, at the link layer several protocols try to reduce the energy spent for idle listening. Therefore, they often switch the radio chip into its sleep state [van Dam and Langendoen 2003; Polastre et al. 2004]. These optimizations are orthogonal to our approach. Energy levels could still be used on other layers, as long as, of course, the same protocol stack is used for both energy profiling and the application itself.

Sensor network simulators like Avrora [Titzer et al. 2005; Landsiedel et al. 2005] and PowerTOSSIM [Shnayder et al. 2004] enable the prediction of the energy consumption of a sensor node. The values obtained from these tools are often used for evaluation purposes and to give the developer hints about energy consumption, although usually not at runtime. Avrora allows to break down energy consumption to individual functions. However, this part of Avrora can only associate the energy consumption of the CPU with some code rather than including the other hardware components on a node. In addition, unlike our approach, it does not take into account the energy consumed by functions that are called by the code under measurement or, in the case of TinyOS, by asynchronously executed tasks.

There are several more advanced battery models than ours described in the literature [Rao et al. 2003]. They take into account effects resulting from temperature changes and time-varying loads, for example. However, because the voltage sensor on typical sensor nodes cannot provide the precision of lab equipment, we have to deal with inaccuracies anyway. Furthermore, the computational overhead of many accurate battery models is too large for resource-constrained sensor nodes, and it takes even for more powerful computers hours to simulate a load profile.

6.6. Summary

In this chapter we have described and evaluated *Levels*, the part of our cross-layer framework that deals with the energy limitations of the sensor nodes. Unlike most approaches that try to maximize network lifetime, *Levels* helps to meet user-defined lifetime goals for a sensor network and for each of its individual nodes. *Levels* requires only small modifications to existing code and its energy levels provide a flexible and easy-to-use programming abstraction. With only minimal extensions to nesC *Levels* allows to mark code that is not needed to provide some basic functionality like network connectivity or sampling with less energy-intensive sensors. In addition, measuring the energy consumption of parts of an application is easy with our simulation-based approach for energy profiling. Finally, our runtime system shields the application developer completely from low-level issues related to lifetime estimation. Even balancing energy levels and node schedules among neighboring nodes for constant application quality can be performed by the system without any effort for the developer.

If an accurate battery model and information about the energy consumption of the sensor nodes are available, *Levels* helps to ensure that each node meets its lifetime goal and provides an application quality that is close to the optimum. Therefore, the developer does not have to deal manually with energy limitations. This will probably make some cross-layer interactions unnecessary.

Levels assumes that future energy consumption can be predicted from information about the past. Although we expect this assumption to be true for the long periods between level adjustments, *Levels* might not be able, however, to meet the lifetime goal in all cases if the node's load changes significantly over time.

Using our coordination algorithm, nodes can balance their energy level assignments and – in dense networks with redundant nodes – their activation schedules in order to reduce fluctuations in application quality. This approach is run in a completely distributed way and requires only minimal data exchange among nodes. As we have shown in the evaluation, the energy overhead of both this coordination and the local optimization is negligible.

In conclusion, we expect that *Levels* will help to make the creation of energy-aware sensor network applications much easier. For applications that cannot benefit from redundant nodes it will allow to preserve some minimal functionality for the lifetime defined by the user. Although this might somewhat decrease the quality of the data obtained from the network, we argue that – especially in a sparse network topology – a node is more useful when providing reduced functionality than if it stops working completely. Similarly, for applications with redundant nodes, *Levels* can help to achieve a constant application quality over the complete network lifetime by computing both the activation schedules and energy level assignments.

7. Summary and Outlook

This chapter summarizes the results from the previous ones and gives a brief outlook on possible future research directions.

7.1. Conclusions

Cross-layer interactions have proved to be an invaluable tool when creating complex applications for sensor networks. Therefore, one should not only consider the negative side effects when thinking about them. However, they definitely raise the complexity for the developer and increase the maintenance overhead. Thus, despite their obvious benefits, they have to be applied carefully.

The main reasons for cross-layer interactions are special properties of wireless communication, the memory limitations of sensor nodes, and their severe energy limitations. Unlike other approaches, our framework addresses all of them.

By analyzing existing applications we have shown that cross-layer interactions are not just a theoretic concept but that they are widely used in practice. Based on this analysis we have created a taxonomy which classifies the different types of cross-layer interactions found.

We are convinced that with appropriate abstractions and system software their negative side-effects can be reduced. Nevertheless, the best solution often would be to do without cross-layer interactions if such a solution can achieve similar performance. Therefore, this thesis has presented a cross-layer framework for sensor networks that pursues different strategies. Its goals are to reduce the negative effects of cross-layer interactions, to address the reason why they are used in the first place, and – by moving some concerns into the system software – to make some of them unnecessary.

First, with *TinyXXL* and the *TinyStateRepository* the framework facilitates data exchange among different layers. As our analysis of existing sensor network applications has shown, this is a form of cross-layer interactions that is frequently used. Optimizations based on data exchange seem to have a significant effect on the efficiency of applications. *TinyXXL* allows to make use of data exchange without coupling the layers more tightly. For example, this data can be used to better deal with the properties of wireless transmission. By performing optimizations at compile-time, *TinyXXL* helps to avoid redundant data and, therefore, potentially reduces memory and energy consumption for applications built from reusable components. Combined with *Neidas*, a neighborhood data sharing algorithm, *TinyXXL* provides a comprehensive system for data exchange both among the layers and software components of a single node and between neighboring nodes. *Neidas* makes use of polite gossiping, i.e., it suppresses transmissions if some neighbors have already sent the request or data, in order to reduce its overhead. Being built upon this concept, it fully leverages the broadcast nature of radio communication.

Second, *ViMem* directly addresses one of the causes of cross-layer interactions. By providing a flash-based virtual memory system it relaxes the memory constraints of the sensor nodes. Therefore, some cross-layer interactions might not be necessary. Using the virtual memory abstraction has the benefit that application developers can access their data independent of its current place in RAM or flash memory. Furthermore, we have adapted this mechanism to the special properties of flash memory and sensor networks. Most important, *ViMem* strives to optimize the memory layout such that the number of energy-expensive accesses to flash memory is reduced.

Finally, *Levels* addresses energy constraints. It allows the application developer to meet a user-defined lifetime goal without having to explicitly deal with the low-level details of energy estimation. Using its fully distributed assignment algorithm *Levels* does not only maximize application quality on each sensor node for the given lifetime, but also ensures a constant overall network quality over time. By providing this functionality as a part of the system software, *Levels* can help to avoid some cross-layer interactions of the application that are caused by energy limitations. Nevertheless, even with *Levels* not all of these cross-layer interactions will become unnecessary.

We have implemented the parts of the cross-layer framework as an extension of TinyOS and nesC for Mica2 nodes. Then we have evaluated them using simulation and experiments with real sensor nodes. By developing or modifying complex applications we have shown that our abstractions, algorithms, and system software can provide benefit to real-world applications. For example, *TinyXXL* and *ViMem* have been applied to TinyDB and *Levels* has been used to adapt an application that monitors volcanic eruptions.

Although there is some runtime overhead associated with the framework, it is almost always negligible. One design principle that helps to keep the overhead low is that as much processing as possible is moved to compile-time. Therefore, the sensor nodes themselves do not have to, for example, select a component that provides some data with *TinyXXL*, optimize the memory layout for *ViMem*, or measure energy consumption of code blocks for *Levels*.

Another design principle that helps to reduce the overall overhead is our modular architecture. Each of the cross-layer framework's three parts can be used individually in applications. Thus only the functionality that is actually needed has to be present on the sensor nodes. In this respect our cross-layer framework pursues the same approach as TinyOS, which also includes only those components in the code image that are actually needed by the application.

In conclusion, the resulting framework addresses a broad range of cross-layer interactions. In fact, in many aspects it is more general than the original target: Some of its abstractions and algorithms are well beyond the traditional scope of cross-layer interactions. For example, *Neidas* can be used not just for typical cross-layer data like link qualities or routing information but also for any other kind of data useful to neighboring nodes. Similarly, *Levels* is not just a tool that helps to avoid some cross-layer interactions but provides powerful mechanisms for energy-aware applications. Therefore, our framework does not simply reduce the negative effects of cross-layer interactions; in general, it rather helps the application developer to create efficient sensor network applications.

7.2. Outlook

Regarding possible future research problems, there are several ways to extend the work of this thesis. First, the set of cross-layer interactions that are dealt with by the framework could be expanded. Second, there are some opportunities to extend the work on each of its parts. Finally, depending on the direction of future sensor network research, some assumptions and trade-offs might have to be reconsidered. This section gives some examples for each of these items.

First of all, even though our framework offers solutions for the most commonly used cross-layer interactions, it does not support all examples for them that we have found in existing applications. For instance, in Chapter 4 some of the examples for cross-layer interactions based on function calls are not supported by our framework. We did not consider these interactions because they are already well-supported by the mechanisms of nesC. However, it could be worthwhile to reevaluate this way of interaction and provide some mechanisms that couple these components less tightly.

Although we have modified existing protocols and developed a real-world application from scratch using our data exchange abstractions, these interactions focus mostly on specific pairs of protocols. However, it would be interesting to see what kinds of cross-layer interactions emerge if a complete network stack with a large number of alternative protocols is developed using *TinyXXL*.

For *Neidas* a possible extension is to consider not just the nodes in radio range but, for example, arbitrary groups of nodes. Although such systems have already been proposed in the literature, they often lack an efficient data sharing algorithm.

Regarding *ViMem* it might be interesting to study the trade-offs if it is applied to different classes of devices that, for example, are equipped with another kind of secondary storage, possibly one of the new memory types currently being developed (e.g., FeRAM).

For *Levels* one important research direction would be to combine it with a protocol that preserves coverage or network connectivity. Furthermore, besides minimizing the deviation from the average, constraints that actually span several nodes could be introduced. This way, for example, the system could ensure that always a given number of nodes provides a predefined energy level. In addition, instead of maximizing application quality for a given lifetime, it could be worthwhile to examine what is different for applications that want to achieve a given quality while maximizing their lifetime. Similarly, the effects of energy harvesting – with the possibility of perpetual applications – should be studied separately in more detail.

Furthermore, it is still an open question how other properties that are considered by TinyCubus – for example, the delivery ratio – can be combined with the abstraction of an energy level.

Depending on the direction of future sensor networks research, different aspects of our work will have to be reconsidered. For example, if true low-power applications emerge that sleep for several weeks or months before becoming active for just a few minutes, some abstractions and algorithms of our framework will have to be adjusted. Most notably, *Neidas* would need to be changed for such a "mostly-off" application since it assumes a broadcast communication mechanism for periodic updates.

If, however, future sensor networks consist of more heterogeneous devices – some of them possibly with more energy and other resources available – some trade-offs in other parts of our cross-layer framework will have to be rethought. For example, if more energy is available, *Levels* and *ViMem* will require some changes to adjust their optimization goals and to make use of that additional energy.

Nevertheless, independent of the direction of sensor network research, we are convinced that in the presence of cheap, resource-constrained devices and wireless communication cross-layer interactions will continue to be an important topic. Furthermore, if such interactions are used in the development of complex commercial applications, system support with accompanying programming abstractions is necessary to create efficient applications at low development costs.

A. Extensions of the nesC Grammar

For reference, this appendix summarizes the changes to the nesC grammar required by our cross-layer framework. In particular, nesC has been extended for *TinyXXL* and *Levels*. For *ViMem*, however, no changes were necessary since it only operates with attributes that are already part of nesC.

This description extends the nesC and C grammars [Gay et al. 2005; Kernighan and Ritchie 1988] and uses the same syntax as those. Words in a `typewriter font` are keywords that are not replaced. Each line in a rule refers to a separate alternative. If a rule is marked with "also", this rule adds alternatives to a standard nesC rule. Finally, optional symbols are marked with "*opt*".

A.1. Changes for TinyXXL

Since *TinyXXL* introduces several new keywords, there are several changes to the nesC grammar. However, many of them are very small, and some rules are only needed because of the syntax used to display the grammar.

nesC-file: also
  xldata-file
  xlparam-file
  xlvirtual-file

xldata-file:
  `xldata` *identifier* `(` *data-parameter-list*$_{opt}$ `)` { *xldata-body* }

xlparam-file:
  `xlparam` *identifier* { *xldata-body* }

xlvirtual-file:
  `xlvirtual` *identifier* { *xlvirtual-specification* } *xlvirtual-implementation*

data-parameter-list:
  data-parameter-declaration
  data-parameter-declaration `,` *data-parameter-list*

data-parameter-declaration:
  parameter-declaration `( < )`
  parameter-declaration `( > )`

xldata-body:
  type-specifier identifier `;`
  type-specifier identifier `;` *xldata-body*

xlvirtual-specification:
  uses-provides-data-list

xlvirtual-implementation:
  `implementation` { *translation-unit* }

uses-provides-data-list:
  uses-provides-data
  uses-provides-data-list uses-provides-data

uses-provides-data:
  `uses` *xldata-uses-specification-element-list*
  `provides` *xldata-provides-specification-element-list*

xldata-uses-specification-element-list:
  xldata-uses-specification-element
  { *xldata-uses-specification-elements* }

xldata-uses-specification-elements:
  xldata-uses-specification-element
  xldata-uses-specification-elements xldata-uses-specification-element

xldata-uses-specification-element:
  xldata-type instance-name$_{opt}$ *xldata-uses-cond*$_{opt}$ *xldata-uses-neigh*$_{opt}$ `;`

xldata-uses-cond:
  `(` *xldata-parameter-cond* `)`

xldata-uses-neigh:
  `[ ]`

xldata-provides-specification-element-list:
  xldata-provides-specification-element
  { *xldata-provides-specification-elements* }

xldata-provides-specification-elements:
    xldata-provides-specification-element
    xldata-provides-specification-elements xldata-provides-specification-element

xldata-provides-specification-element:
    xldata-type instance-name$_{opt}$ *xldata-provides-parameter*$_{opt}$ `;`

xldata-provides-parameter:
    `(` *xldata-parameter-specification* `)`

xldata-type:
    `xldata` *identifier*

xldata-parameter-cond:
    expression

xldata-parameter-specification:
    xldata-parameter-list

xldata-parameter-list:
    xldata-parameter-value
    xldata-parameter-list `,` *xldata-parameter-value*

xldata-parameter-value:
    identifier
    constant

statement: also
    data-providing-statement

data-providing-statement:
    `ifproviding (` *identifier* `)` *statement*

storage-class-specifier: also
    `xldata`

specification-element: also
    xldata-specification-element
    xlparam-type instance-name$_{opt}$ `;`

xlparam-type:
    `xlparam` *identifier*

A.2. Changes for Levels

For *Levels*, much fewer changes are needed since it mostly builds on nesC syntax. In fact, just a single rule has to be modified in order to specify energy levels. No changes are necessary for wiring energy levels.

specification-element: also
  `energylevel` *identifier* `<` *integer-constant* `>`

No more modifications are needed because using energy levels is very similar to interfaces. In the implementation of *Levels* the pre-compiler, in fact, replaces them with an interface. Implementing energy levels completely with interfaces is not possible since, otherwise, the local order of energy levels within the module could not be specified.

Bibliography

[Abrach et al. 2003] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: System support for multimodal networks of in-situ sensors. In *Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*, pp. 50–59, 2003.

[Arora et al. 2004] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: a wireless sensor network for target detection, classification, and tracking. *Comput. Networks*, 46(5):605–634, 2004.

[Atmel Corporation 2005] Atmel Corporation. *4-megabit DataFlash AT45DB041B Datasheet*, 2005.

[Beutel et al. 2004] J. Beutel, O. Kasten, F. Mattern, K. Römer, F. Siegemund, and L. Thiele. Prototyping wireless sensor network applications with BTnodes. In *Proceedings of the 1st European Workshop on Sensor Networks (EWSN 2004)*, pp. 323–338, 2004.

[Calder et al. 1998] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 139–149, 1998.

[Camacho and Bordons 2004] E. F. Camacho and C. Bordons. *Model Predictive Control.* Advanced Textbooks in Control and Signal Processing. Springer-Verlag, 2nd edition, 2004.

[Cardei and Wu 2006] M. Cardei and J. Wu. Energy-efficient coverage problems in wireless ad-hoc sensor networks. *Computer Communications*, 29(4):413–420, 2006.

[Cerpa and Estrin 2002] A. Cerpa and D. Estrin. ASCENT: Adaptive self-configuring sensor networks topologies. In *Proc. of the Twenty-First Annual Joint Conf. of the IEEE Computer and Communications Societies*, volume 3, pp. 1278–1287, 2002.

[Cheong et al. 2003] E. Cheong, J. Liebman, J. Liu, and F. Zhao. TinyGALS: A programming model for event-driven embedded systems. In *Proc. of the 2003 ACM Symposium on Applied Computing*, pp. 698–704, 2003.

[Chvátal 1983] V. Chvátal. *Linear Programming.* W. H. Freeman and Company, 1983.

[Conti et al. 2004] M. Conti, G. Maselli, G. Turi, and S. Giodano. Cross-layering in mobile ad hoc network design. *IEEE Computer*, 37(2):48–51, 2004.

[Dai et al. 2004] H. Dai, M. Neufeld, and R. Han. ELF: An efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pp. 176–187, 2004.

[Denning 1970] P. J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, 1970.

[Dunkels et al. 2004] A. Dunkels, B. Grönvall, and T. Voigt. Contiki – a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I)*, 2004.

[Dunkels et al. 2006] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pp. 29–42, 2006.

[Dunkels et al. 2007] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based on-line energy estimation for sensor nodes. In *Proc. of the Fourth Workshop on Embedded Networked Sensors*, 2007.

[Duracell Batteries] Duracell Batteries. Duracell Plus alkaline-manganese dioxide battery. `http://www.mdsbattery.co.uk/datasheets/duracell/MN1500PL.pdf`.

[Dutta and Culler 2005] P. K. Dutta and D. E. Culler. System software techniques for low-power operation in wireless sensor networks. In *Proc. of the 2005 International Conference on Computer-Aided Design*, 2005.

[Estrin et al. 1999] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pp. 263–270. ACM Press, 1999.

[Flinn and Satyanarayanan 1999] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proc. of the Seventeenth ACM Symposium on Operating Systems Principles*, pp. 48–63, 1999.

[Franklin and Zdonik 1997] M. Franklin and S. Zdonik. A framework for scalable dissemination-based systems. In *Proc. of the 12th Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 94–105, 1997.

[Gal and Toledo 2005] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005.

[Garlan and Shaw 1993] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pp. 1–39, 1993.

[Gay et al. 2003] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the Conf. on Programming Language Design and Implementation*, pp. 1–11, 2003.

[Gay et al. 2005] D. Gay, P. Levis, D. Culler, and E. Brewer. *nesC 1.2 Language Reference Manual*, 2005.

[Gehrke and Madden 2004] J. Gehrke and S. Madden. Query processing in sensor networks. *Pervasive Computing*, 3(1):46–55, 2004.

[Giusti et al. 2007] A. Giusti, A. L. Murphy, and G. P. Picco. Decentralized scattering of wake-up times in wireless sensor networks. In *Proc. of the 4th European Conference on Wireless Sensor Networks*, pp. 245–260, 2007.

[Goldsmith and Wicker 2002] A. J. Goldsmith and S. B. Wicker. Design challenges for energy-constrained ad hoc wireless networks. *IEEE Wireless Communications*, 9(4):8–27, 2002.

[Greenstein et al. 2004] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (SNACK). In *Proc. of the 2nd International Conference on Embedded Networked Sensor Systems*, pp. 69–80, 2004.

[Gu and Stankovic 2006] L. Gu and J. A. Stankovic. t-kernel: Providing reliable OS support to wireless sensor networks. In *Proc. of the 4th International Conference on Embedded Networked Sensor Systems*, pp. 1–14, 2006.

[Gupta 1991] R. Gupta. *Compiler Optimization of Data Storage.* PhD thesis, California Institute of Technology, 1991.

[Han et al. 2005] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, pp. 163–176, 2005.

[Hartley 1988] S. J. Hartley. Compile-time program restructuring in multiprogrammed virtual memory systems. *IEEE Trans. Softw. Eng.*, 14(11):1640–1644, 1988.

[Hatfield and Gerald 1971] D. J. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[Heinzelman et al. 1999] W. R. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pp. 174–185, 1999.

[Hill et al. 2000] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of the 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 93–104, 2000.

[Hill et al. 2004] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy. The platforms enabling wireless sensor networks. *Communications of the ACM*, 47(6):41–46, 2004.

[Huang et al. 2006] C.-F. Huang, L.-C. Lo, Y.-C. Tseng, and W.-T. Chen. Decentralized energy-conserving and coverage-preserving protocols for wireless sensor networks. *ACM Trans. Sen. Netw.*, 2(2):182–187, 2006.

[Hui and Culler 2004] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of the 2nd Int'l Conf. on Embedded Networked Sensor Systems*, pp. 81–94, 2004.

[Intanagonwiwat et al. 2000] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. of the 6th Annual International Conference on Mobile Computing and Networking*, pp. 56–67, 2000.

[Jiang et al. 2007a] X. Jiang, P. Dutta, D. Culler, and I. Stoica. Micro power meter for energy monitoring of wireless sensor networks at scale. In *Proc. of the 6th Int'l Conf. on Information Processing in Sensor Networks: Track on Sensor Platforms, Tools and Design Methods*, pp. 186–195, 2007a.

[Jiang et al. 2007b] X. Jiang, J. Taneja, P. D. Jorge Ortiz and, Arsalan Tavakoli and, J. Jeong, D. Culler, P. Levis, and S. Shenker. An architecture for energy management in wireless sensor networks. In *Proc. of the International Workshop on Wireless Sensor Network Architecture*, 2007b.

[Juang et al. 2002] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with ZebraNet. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 96–107, 2002.

[JUnit] JUnit. JUnit web site. `http://www.junit.org`.

[Karp and Kung 2000] B. Karp and H. T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proc. of the 6th Annual International Conference on Mobile Computing and Networking*, pp. 243–254, 2000.

[Kawadia and Kumar 2005] V. Kawadia and P. R. Kumar. A cautionary perspective on cross layer design. *IEEE Wireless Communications*, 12(1):3–11, 2005.

[Kernighan and Ritchie 1988] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[Kim et al. 2007] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *Proc. of the 6th International Conference on Information Processing in Sensor Networks*, pp. 254–263, 2007.

[Kirkpatrick et al. 1983] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[Koshy and Pandey 2005] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proc. of the 2nd European Workshop on Wireless Sensor Networks*, pp. 354–365, 2005.

[Köpke et al. 2004] A. Köpke, V. Handziski, J.-H. Hauer, and H. Karl. Structuring the information flow in component-based protocol implementations for wireless sensor nodes. In *Proc. of Work-in-Progress Session of the 1st European Workshop on Wireless Sensor Networks (EWSN)*, Technical Report TKN-04-001 of Technical University Berlin, Telecommunication Networks Group, pp. 41–45, 2004.

[Krishnamurthy et al. 2005] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and deployment of industrial sensor networks: Experiences from a semiconductor plant and the north sea. In *Proc. of the 3rd International Conference on Embedded Networked Sensor Systems*, pp. 64–75, 2005.

[Kumar et al. 2006] R. Kumar, S. PalChaudhuri, C. Reiss, and U. Ramachandran. System support for cross-layering in sensor network stack. In *Proc. of the 2nd International Conference on Mobile Ad-hoc and Sensor Networks*, pp. 793–807, 2006.

[Lachenmann et al.] A. Lachenmann, K. Herrmann, K. Rothermel, and P. J. Marrón. On meeting lifetime goals and providing constant application quality. In preparation.

[Lachenmann et al. 2005] A. Lachenmann, P. J. Marrón, D. Minder, and K. Rothermel. An analysis of cross-layer interactions in sensor network applications. In *Proc. of the Second International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, pp. 121–126, 2005.

[Lachenmann et al. 2006] A. Lachenmann, P. J. Marrón, D. Minder, M. Gauger, O. Saukh, and K. Rothermel. TinyXXL: Language and runtime support for cross-layer interactions. In *Proc. of the Third Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, pp. 178–187, 2006.

[Lachenmann et al. 2007a] A. Lachenmann, P. J. Marrón, M. Gauger, D. Minder, O. Saukh, and K. Rothermel. Removing the memory limitations of sensor networks with flash-based virtual memory. In *Proc. of the European Conference on Computer Systems (EuroSys)*, pp. 131–144, 2007a. Also published in ACM SIGOPS Operating Systems Review, vol. 41(3), 2007.

[Lachenmann et al. 2007b] A. Lachenmann, P. J. Marrón, D. Minder, and K. Rothermel. Meeting lifetime goals with energy levels. In *Proc. of the 5th ACM Conference on Embedded Networked Sensor Systems*, pp. 131–144, 2007b.

[Lachenmann et al. 2007c] A. Lachenmann, P. J. Marrón, D. Minder, O. Saukh, M. Gauger, and K. Rothermel. Versatile support for efficient neighborhood data sharing. In *Proceedings of the Fourth European Conference on Wireless Sensor Networks (EWSN 2007)*, pp. 1–16, 2007c.

[Landsiedel et al. 2005] O. Landsiedel, K. Wehrle, and S. Götz. Accurate prediction of power consumption in sensor networks. In *Proc. of the Second Workshop on Embedded Networked Sensors*, 2005.

[Langendoen et al. 2006] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *Proc. of the International Parallel and Distributed Processing Symposium*, pp. 8–15, 2006.

[Levis et al. 2004a] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in TinyOS. In *Proc. of the 1st Symposium on Network Systems Design and Implementation*, 2004a.

[Levis et al. 2004b] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. of the 1st Symp. on Networked Systems Design and Implementation*, pp. 15–28, 2004b.

[Levis et al. 2005] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proceedings of the 2nd Symp. on Networked Systems Design and Implementation (NSDI)*, 2005.

[Liu et al. 2004] T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi. Implementing software on resource-constrained mobile sensors: Experiences with Impala and ZebraNet. In *Proc. of the International Conference on Mobile Systems, Applications, and Services*, pp. 256–269, 2004.

[Madden et al. 2002] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[Madden et al. 2005] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[Marrón et al. 2005a] P. J. Marrón, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel. TinyCubus: A flexible and adaptive framework for sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, pp. 278–289, 2005a.

[Marrón et al. 2005b] P. J. Marrón, D. Minder, A. Lachenmann, and K. Rothermel. TinyCubus: An adaptive cross-layer framework for sensor networks. *it – Information Technology*, 47(2):87–97, 2005b.

[Marrón et al. 2005c] P. J. Marrón, O. Saukh, M. Krüger, and C. Große. Sensor network issues in the Sustainable Bridges project. In *European Projects Session of EWSN 2005*, 2005c.

[Marrón et al. 2006a] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. FlexCup: A flexible and efficient code update mechanism for sensor networks. In *Proc. of the Third European Workshop on Wireless Sensor Networks*, pp. 212–227, 2006a.

[Marrón et al. 2006b] P. J. Marrón, D. Minder, and the Embedded WiSeNts Consortium, editors. *Embedded WiSeNts Research Roadmap.* Logos, 2006b.

[Maté] Maté. Maté web page. `http://www.cs.berkeley.edu/~pal/mate-web/`.

[Melodia et al. 2005] T. Melodia, M. C. Vuran, and D. Pompili. The State of the Art in Cross-layer Design for Wireless Sensor Networks. In *Proceedings of the Second International Workshop of the EURO-NGI Network of Excellence*, 2005.

[Minder et al. 2005] D. Minder, P. J. Marrón, A. Lachenmann, and K. Rothermel. Experimental construction of a meeting model for smart office environments. In *Proceedings of the First Workshop on Real-World Wireless Sensor Networks (REALWSN 2005), SICS Technical Report T2005:09*, 2005.

[Mottola and Picco 2006] L. Mottola and G. P. Picco. Logical neighborhoods: A programming abstraction for wireless sensor networks. In *Proc. of the Int'l Conf. on Distributed Computing in Sensor Systems*, pp. 150–168, 2006.

[Muchnick 1997] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Academic Press, 1997.

[Nii 1986] P. Nii. The blackboard model of problem solving. *AI Magazine*, 7(2):38–53, 1986.

[Ostermann 2007] A. Ostermann. Verteilte Bestimmung von Energiestufen in drahtlosen Sensornetzen. Diploma thesis, Universität Stuttgart, 2007.

[Polastre et al. 2004] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proc. of the Int'l Conf. on Embedded Networked Sensor Systems*, pp. 95–107, 2004.

[Polastre et al. 2005a] J. Polastre, J. Hui, P. Levis, J. Yhao, D. Culler, S. Shenker, and I. Stoica. A unifying link abstraction for wireless sensor networks. In *Proc. of the 3rd International Conference on Embedded Networked Sensor Systems*, pp. 76–89, 2005a.

[Polastre et al. 2005b] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. of the Int'l Conf. on Information Processing in Sensor Networks: Track on Sensor Platform, Tools and Design Methods for Networked Embedded Systems*, 2005b.

[Raisinghani and Iyer 2004] V. T. Raisinghani and S. Iyer. Cross-layer design optimizations in wireless protocol stacks. *Elsevier Computer Communications*, 27(8):720–724, 2004.

[Rao et al. 2003] R. Rao, S. Vrudhula, and D. N. Rakhmatov. Battery modeling for energy-aware system design. *Computer*, 36(12):77–87, 2003.

[Reichardt et al. 2002] D. Reichardt, M. Miglietta, L. Moretti, P. Morsink, and W. Schulz. CarTALK 2000: Safe and comfortable driving based upon inter-vehicle-communication. In *Proc. of the Intelligent Vehicle Symp.*, volume 2, pp. 545–550, 2002.

[Römer and Mattern 2004] K. Römer and F. Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11:54–61, 2004.

[Römer et al. 2004] K. Römer, C. Frank, P. J. Marrón, and C. Becker. Generic role assignment for wireless sensor networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, pp. 7–12, 2004.

[Sallai et al. 2004] J. Sallai, G. Balough, M. Maróti, and Ákos Lédeczi. Acoustic ranging in resource constrained sensor networks. Technical Report ISIS-04-504, Vanderbilt University, 2004.

[Shakkottai et al. 2003] S. Shakkottai, T. S. Rappaport, and P. C. Karlsson. Cross-layer design for wireless networks. *IEEE Communications Magazine*, 41(10):74–80, 2003.

[Shnayder et al. 2004] V. Shnayder, M. Hempstead, B.-r. Chen, G. Werner-Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. of the 2nd Int'l Conf. on Embedded Networked Sensor Systems*, pp. 188–200, 2004.

[Silberschatz et al. 2002] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts.* John Wiley & Sons, 6th edition, 2002.

[Sommerville 2001] I. Sommerville. *Software Engineering.* Addison-Wesley, 6th edition, 2001.

[Sorber et al. 2007] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: A language and runtime system for perpetual systems. In *Proc. of the 5th International Conference on Embedded Networked Sensor Systems*, pp. 161–174, 2007.

[Srivastava and Motani 2005] V. Srivastava and M. Motani. Cross-layer design: A survey and the road ahead. *IEEE Communications Magazine*, 43(12):112–119, 2005.

[Stamos 1984] J. W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Trans. Comput. Syst.*, 2(2):155–180, 1984.

[Su and Lim 2006] W. Su and T. L. Lim. Cross-layer design and optimization for wireless sensor networks. In *Proc. of the Int'l Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pp. 278–284, 2006.

[Szewczyk et al. 2004] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin. Habitat monitoring with sensor networks. *Comm. of the ACM*, 47(6):34–40, 2004.

[Tanenbaum 2003] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, 4th edition, 2003.

[Tian et al. 2003] J. Tian, L. Han, K. Rothermel, and C. Cseh. Spatially aware packet routing for mobile ad hoc inter-vehicle radio networks. In *Proc. of the IEEE 6th Intl. Conf. on Intelligent Transportation Systems (ITSC)*, volume 2, pp. 1546–1551, 2003.

[TinyOS] TinyOS. TinyOS CVS repository. `http://tinyos.cvs.sourceforge.net/`.

[Titzer and Palsberg 2005] B. Titzer and J. Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *Proc. of the Conf. on Languages, Compilers, and Tools for Embedded Systems*, pp. 59–68, 2005.

[Titzer et al. 2005] B. Titzer, D. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of the Fourth Int'l Conf. on Information Processing in Sensor Networks*, pp. 477–482, 2005.

[Tolle and Culler 2005] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proc. of the Second European Workshop on Wireless Sensor Networks*, pp. 121–132, 2005.

[Tolle et al. 2005] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *Proc. of the 3rd International Conference on Embedded Networked Sensor Systems*, pp. 51–63, 2005.

[van Dam and Langendoen 2003] T. van Dam and K. Langendoen. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *Proc. of the Int'l Conf. on Embedded Networked Sensor Systems*, pp. 171–180, 2003.

[van Hoesel et al. 2004] L. van Hoesel, T. Nieberg, J. Wu, and P. J. M. Havinga. Prolonging the lifetime of wireless sensor networks by cross-layer interaction. *IEEE Wireless Communications*, 11(6):78–86, 2004.

[Wark et al. 2007] T. Wark, P. Corke, P. Sikka, L. Klingbeil, Y. Guo, C. Crossman, P. Valencia, D. Swain, and G. Bishop-Hurley. Transforming agriculture through pervasive wireless sensor networks. *IEEE Pervasive Computing*, 6(2):50–57, 2007.

[Weinschrott 2007] H. Weinschrott. Protocol to acquire and cache large data in sensor networks. Diploma thesis, Universität Stuttgart, 2007.

[Welsh and Mainland 2004] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proc. of the 1st Symp. on Networked Systems Design and Implementation*, pp. 29–42, 2004.

[Werner-Allen et al. 2006a] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. of the Symp. on Operating Systems Design and Implementation*, 2006a.

[Werner-Allen et al. 2006b] G. Werner-Allen, K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, and J. Lees. Deploying a wireless sensor network on an active volcano. *IEEE Internet Computing*, 10(2):18–25, 2006b.

[Whitehouse et al. 2004] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proc. of the 2nd International Conference on Mobile Systems, Applications, and Services*, pp. 99–110, 2004.

[Xu et al. 2004] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Proc. of the 2nd International Conference on Embedded Networked Sensor Systems*, pp. 13–24, 2004.

[Yavatkar et al. 1995] R. Yavatkar, J. Griffoen, and M. Sudan. A reliable dissemination protocol for interactive collaborative applications. In *Proc. of the Third ACM International Conference on Multimedia*, pp. 333–344, 1995.

[Ye et al. 2002] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proc. of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, pp. 1567–1576, 2002.

[Ye et al. 2003] F. Ye, G. Zhong, J. Cheng, S. Lu, and L. Zhang. PEAS: A robust energy conserving protocol for long-lived sensor networks. In *Proc. of the Int'l Conf. on Distributed Computing Systems*, pp. 28–37, 2003.

[Zeng et al. 2002] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 123–132, 2002.

Index