



universität**bonn**

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik IV: Kommunikation und Verteilte Systeme
Arbeitsgruppe Sensornetze und Pervasive Computing

Abschlussvortrag zur SNPC Diplomarbeit 02

Logging Tree System (LTS)

Entwurf und Implementierung eines effizienten verteilten Logging- und Tracing-Frameworks für TinyOS

Oliver Frietsch
frietsch@cs.uni-bonn.de

Überblick

- **Sensorknoten (Mote)**
 - Preiswerter Kleinstrechner mit Umweltsensoren und Drahtlos-Netzwerkadapter
 - Sehr geringe Rechenleistung und Speicherausstattung
- **Sensornetz**
 - Netz aus vielen Sensorknoten
 - Häufig ohne aufwändige Administration betrieben
- **Software-Entwicklung in Sensornetzen: Schwierig!**
 - Eingeschränkte Ressourcen → Effiziente Software erforderlich
 - Verteiltes System mit vielen Akteuren } Viele
 - Unzuverlässiger Betrieb des Netzwerks } Unwägbarkeiten!
- **Großes Problem: Fehlersuche**
 - Effiziente und einfach zu nutzende Hilfsmittel erforderlich
 - Ziel: Generisches Logging- und Tracing-Framework

Inhalt

- Grundlagen
- Ziele
- Logging Tree System (LTS)
 - Anwendung
 - Aufbau und Umsetzung
 - Evaluation
- Zusammenfassung
- Ausblick
- Fragen

- **Komponenten- und ereignisorientierter C-Dialekt**
 - Erlaubt den Aufbau komplexer Programme aus kompakten Komponenten
 - Single-Threaded, sequenzielle Behandlung von Ereignissen
- **Interfaces**
 - Deklarieren thematisch verwandte öffentliche Operationen (Commands/Events)
 - Keine Implementierung!
- **Komponenten**
 - Bieten Interfaces an oder setzen sie voraus
 - Zwei Gruppen
 - Module: Ausführbarer Code, implementieren Interfaces
 - Configurations: Verbinden mehrere Komponenten zu einer gemeinsamen Komponente
 - Dabei sind Interfaces durch „Wirings“ miteinander zu verbinden
 - Meist singleton, also nur eine Instanz (mit Ausnahme generischer Komponenten)
- **Attribute**
 - nesC-Notation zur Angabe semantischer Informationen im Quelltext
 - Nutzbar z.B. an Operations-, Variablen- und Typdeklarationen

- Sensorknoten sind nicht leistungsfähig genug für handelsübliche Betriebssysteme
 - Betriebssystem als Abstraktionsebene dennoch wünschenswert
- TinyOS: „Betriebssystem“ für Sensorknoten diverser Plattformen
 - Sammlung aus nesC-Komponenten und -Interfaces, die gemeinsam übliche Betriebssystemfunktionalität bereitstellen
 - Keine isolierte Installation auf dem Sensorknoten
 - Nur von einer Anwendung benötigte TinyOS-Komponenten werden gemeinsam mit der Anwendung in ein monolithisches Software-Image vereint
 - TinyOS Build-System erleichtert Build und Installation
 - Active Messaging Netzwerk-Stack
 - Single Hop über Netzwerkadapter oder auch serielle Schnittstelle
 - Keine eingebaute Routing-Schicht!
 - Jeder Knoten besitzt eine eindeutige Id, gleichzeitig seine Netzwerkadresse
 - Paralleler Betrieb mehrerer logischer Kanäle (Protokolle) möglich

Entwicklung neuer Anwendungen (Phasen)

- **Simulation bzw. Emulation**
 - Virtuelles Sensornetz bzw. Netz aus größeren Rechnern
 - Kostengünstig
 - Einfach zu überwachen
 - Unter Umständen unzureichender Realismus
- **Testbed**
 - Reales Sensornetz aus wenigen universellen Knoten
 - Häufig verkabelt (Stromversorgung, Netzwerk, serielle Schnittstellen, usw.)
- **Deployment**
 - Reales Sensornetz aus speziellen Sensorknoten
 - Meist kabellos (Batteriebetrieb!)
 - Sensorknoten oft nur noch schlecht erreichbar oder sogar mobil

Fehlersuche / Debugging

- **Step-by-Step-Debugging**
 - Anhalten einzelner Sensorknoten
 - Problem: Netz besteht aus interagierenden Systeme mit teilweise zeitkritischen Kommunikations-Protokollen
 - Anhalten einzelner Knoten verändert Verhalten!
 - Gesamtes Netz lässt sich nicht absolut synchron anhalten!
- **Protokollierung von Ereignissen**
 - Ereignis: Einzelner Zeitpunkt des Programmablaufs
 - Passive Beobachtung von Zuständen bei Ereignissen
 - Weniger Eingriffe in den Ablauf → Weniger Seiteneffekte auf das Verhalten!
 - Logging
 - Allgemeiner Begriff für Protokollierung
 - Herkunft: Holzscheit (Log), Frühes Medium zum „Anschreiben“
 - Tracing
 - Verfolgung des Programmablaufs
 - Protokollierung von Operationsaufrufen

Bestehende Systeme für TinyOS

- **Step-by-Step Debugging**
 - Simulatoren (TOSSIM) bzw. Emulatoren (Avrora) über GNU gdb
 - Evtl. nicht realistisch genug!
 - JTAG mit entsprechendem Adapterkabel und z.B. GNU gdb
 - Zu viele Seiteneffekte!
 - Clairvoyant über Drahtlos-Netzwerk (Flooding-Protokoll)
 - Funktionsumfang ähnlich wie GNU gdb
 - Zu viele Seiteneffekte!
- **Logging und Tracing**
 - TOSSIM dbg
 - Ähnlich zu printf in C
 - Nur innerhalb der Simulation mit TOSSIM verfügbar
 - TinyOS printf
 - printf über serielle Schnittstelle
 - Setzt Verkabelung aller einzelnen Knoten voraus
 - In großen Netzen kaum sinnvoll

Ziele

- Generisches Logging- und Tracing-Framework
 - Komfortable Spezifikation der Loggings und Tracings
 - Keine Verdrahtung jedes einzelnen Knotens
 - Intelligenter Betrieb über das Drahtlos-Netzwerk
 - Minimierung der übertragenen Datenmenge
 - Vermeidung von Interferenzen mit anderen Netzwerkprotokollen
 - Ausgabe der Protokollmeldungen auf dem PC

Logging Tree System (LTS)

- Generisches Logging- und Tracing-Framework
Logging Tree System (LTS)
 - Komfortable Spezifikation der Loggings und Tracings
LTS explizite Protokollierungen und Annotationen
LTS nesC-Compiler: Transformation in ausführbaren Code
 - Keine Verdrahtung jedes einzelnen Knotens
 - Intelligenter Betrieb über das Drahtlos-Netzwerk
LTS Logging Tree Protocol (LTP): Priorisierung, Fragmentierung
 - Minimierung der übertragenen Datenmenge
LTS nesC-Compiler: Extraktion statischer Daten
LTS LTP: Beschränkung auf dynamische Daten, Paketzusammenfassung
 - Vermeidung von Interferenzen mit anderen Netzwerkprotokollen
LTS LTP: FreeAirtimeEstimator
 - Ausgabe der Protokollmeldungen auf dem PC
LTS Log Viewer

Spezifikation Loggings und Tracings

```
typedef struct demo_t {  
    char someText[10];  
    uint8_t textIndex;  
} demo_t;
```

- **Tracing: Annotation**

- **@logInvocation(Gewicht, Formatierungs-String, Parameterbezeichner)**
 - Mögliche Gewichte: Error, Warn, Info, Debug
 - Formatierung: Analog zu printf, einige zusätzliche Platzhalter

```
#include <Lts.h>  
command void SomeInterface.someCommand(uint8_t x, demo_t *demo)  
    @logInvocation(SEVERITY_INFO, "x = %d, Char in Struct: %s",  
                 "x, demo->someText[(^demo).textIndex]") {  
    .....  
}
```

- **Logging: Explizite Protokollierung**

- **logNow((Gewicht, Formatierungs-String, Variablen...)**

```
#include <Lts.h>  
void someFunction(uint8_t factor) {  
    uint16_t x = 5*factor;  
    logNow((SEVERITY_WARN, "x is %d, because factor is %d", x, factor));  
}
```

Spezifikation Loggings und Tracings

- **Tracing: Annotation**

- **@logInvocation**(Gewicht, Formatierungs-String, Parameterbezeichner)
 - Mögliche Gewichte: Error, Warn, Info, Debug
 - Formatierung: Analog zu printf, einige zusätzliche Platzhalter

```
#include <Lts.h>
command void SomeInterface.someCommand(uint8 t x, demo t *demo)
    @logInvocation(SEVERITY_INFO, "x = %d, Char in Struct: %s",
                  "x, demo->someText[ (^demo).textIndex]") {
    .....
}
```

Dynamische Informationen: Nur Parameterwerte/Variablenwerte und Zeitstempel!

- **Logging: Explizite Protokollierung**

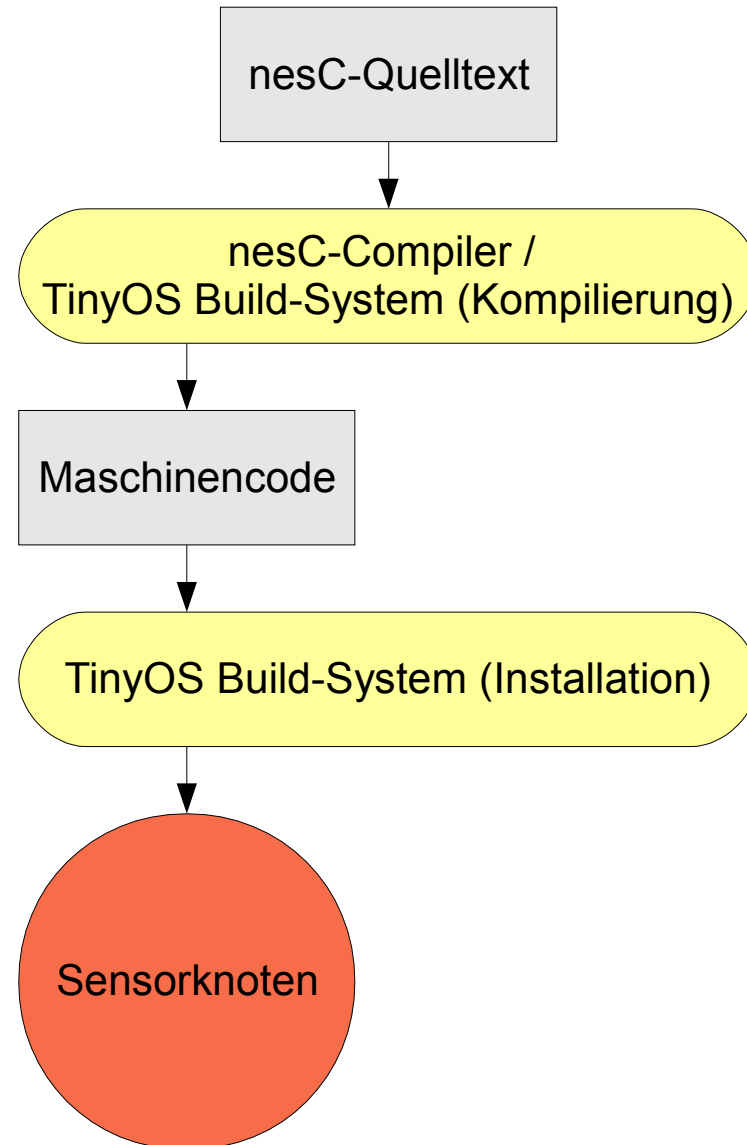
- **logNow**((Gewicht, Formatierungs-String, Variablen...))

```
#include <Lts.h>
void someFunction(uint8 t factor) {
    uint16 t x = 5*factor;
    logNow((SEVERITY_WARN, "x is %d, because factor is %d", x, factor));
}
```

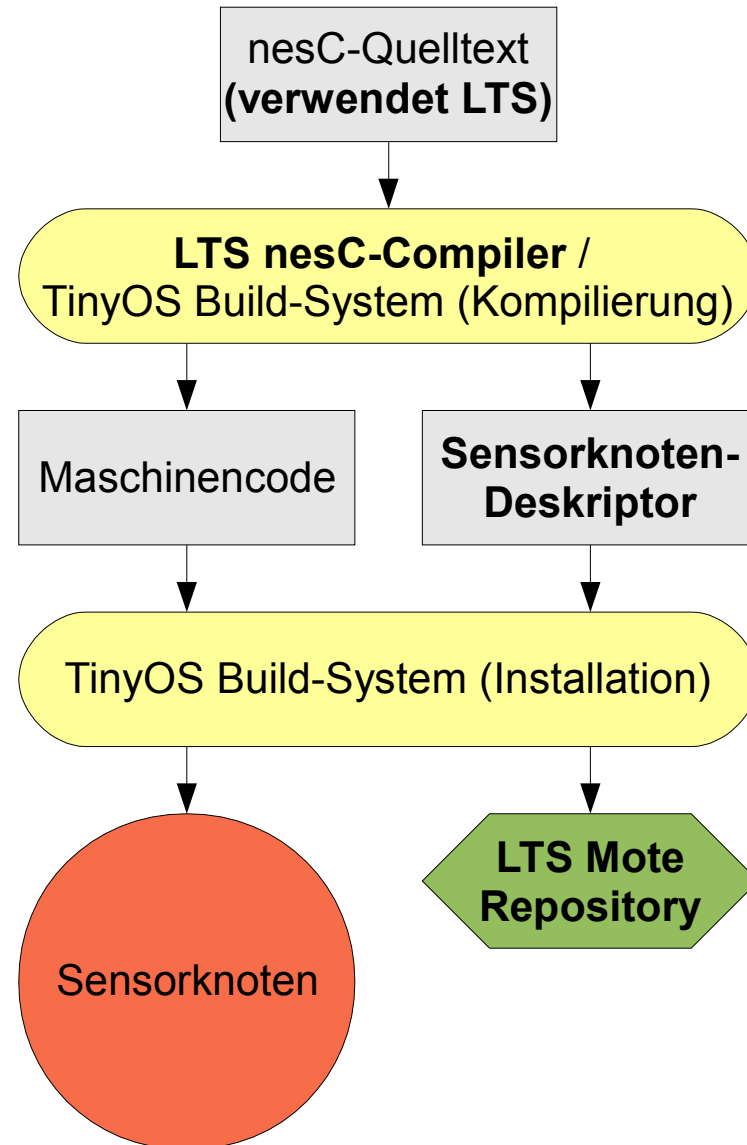
Sensorknoten-Deskriptor

- **Inhalt: Statische Teile der Ereignis/Logging-Meldungen und weitere statische Informationen**
 - Plattformtyp des Sensorknotens
 - Statische Informationen zu allen einzelnen Ereignis-Meldungen
 - Automatisch vergebene Identifikationsnummer (Event Id)
 - Eindeutig auf einem einzelnen Sensorknoten
 - Gewicht
 - Formatierungs-String
 - Name (Operations- und ggf. Interface-Name)
 - Ort (Dateiname und Zeilennummer)
 - Name, Typ und ggf. Größe aller protokollierten Variablen bzw. Parameter
 - Unterstützung für Structs, Felder, Pointer
 - Dateiformat: XML
- **Speicherort: LTS Mote Repository**
 - Zentraler Ordner mit einem Deskriptor pro Sensorknoten
 - Zuordnung über Netzwerkadresse

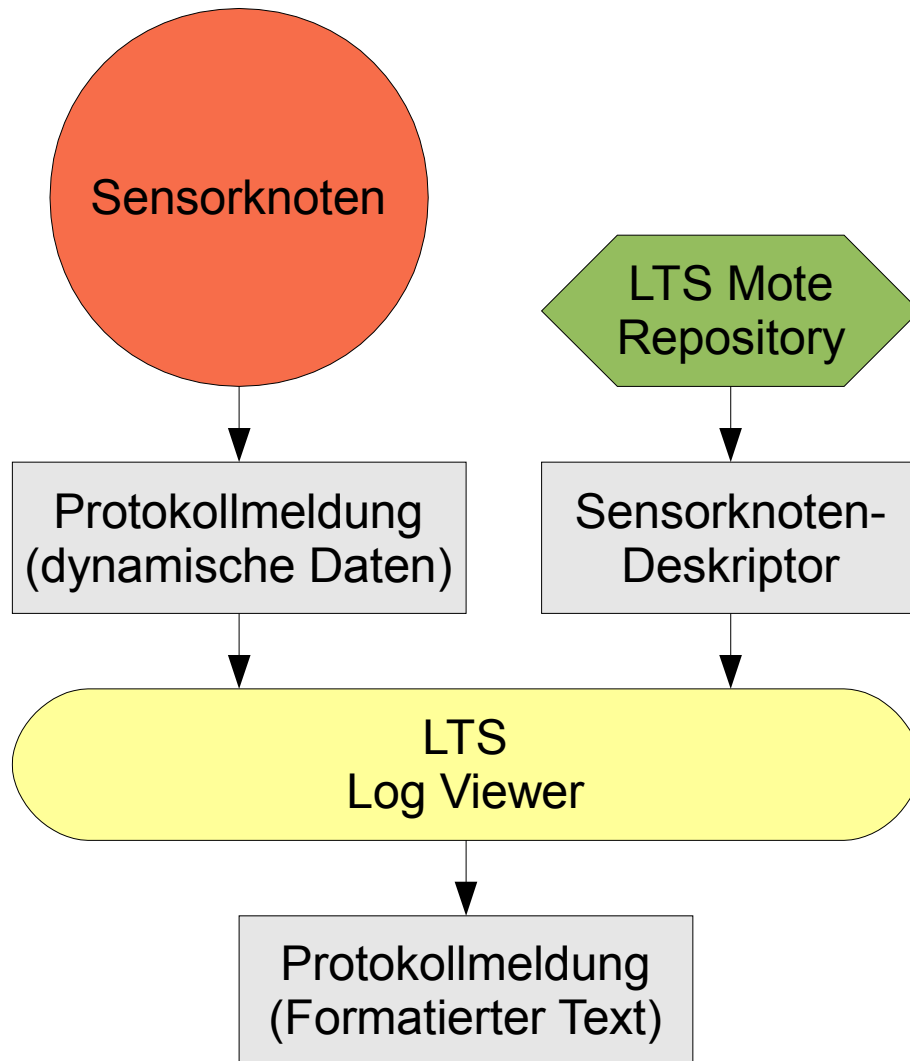
Kompilierung/Installation ohne LTS



Kompilierung/Installation mit LTS



Logging-Betrieb mit LTS

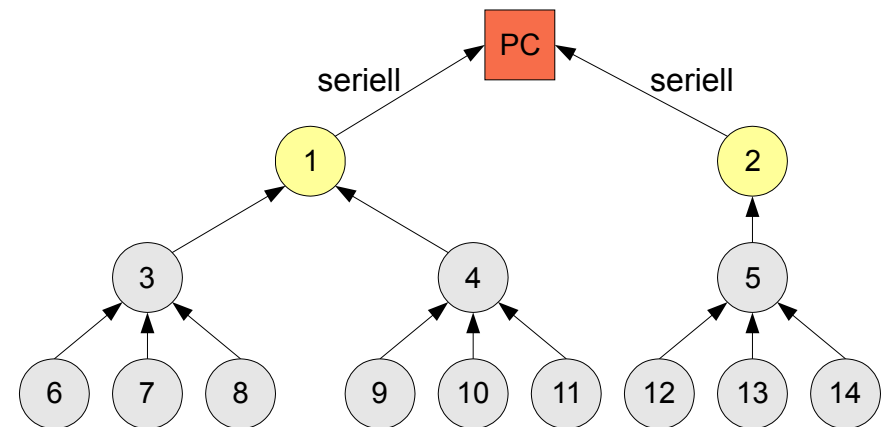


- LTS Log Viewer

- Java-Anwendung
- Kann Daten von beliebig vielen Sensorknoten auslesen
- Fügt dynamische mit statischen Daten zusammen
- Beachtet Unterschiede verschiedener Knoten-Plattformen
 - Typ-Größen
 - Alignment & Padding
 - Design erlaubt Unterstützung zukünftiger Plattformen
- Flexible Architektur
 - Ausgabe auf der Konsole
 - Graphische Benutzeroberfläche möglich

Logging Tree Protocol (LTP)

- **Netzwerkprotokoll innerhalb von LTS, überträgt:**
 - Dynamische Teile der Ereignis/Logging-Meldungen
 - Zeitstempel
 - Ereignisdaten
 - Daten zur Ermittlung der zugehörigen statischen Daten
 - Absenderadresse → Sensorknoten-Deskriptor
 - Event Id → Ereignis-Deskriptor innerhalb des Sensorknoten-Deskriptors
- **Übertragungs-Prinzip: ConvergeCast**
 - Beliebige Anzahl von **Basisstationen**
 - Seriell an PC angebunden
 - Alle Knoten senden ihre Daten
 - Erreichbarer Basisstation oder
 - Erreichbarem Knoten, der näher an einer Basisstation liegt (Elternknoten)
 - Alle Knoten leiten auch empfangene Pakete weiter
 - Aus der Sicht des externen Beobachters bildet sich ein Baum
 - Einzelne Knoten speichern jedoch nur die Adresse ihres Elternknotens!



Collection Tree Protocol (CTP)

- Direkter Bestandteil von TinyOS, implementiert einen ConvergeCast
 - Diente daher bei der Entwicklung von LTP als Basis
- Routing: Adresslos
 - Elternknoten werden ausgewählt anhand der Anzahl der Versuche, die man über sie vermutlich benötigt, um Daten an eine Basisstation zu senden
- Daten
 - Forwarding Engine mit Paket-Warteschlange für Versand
 - Überlaufmeldung mittels „Congestion“-Nachrichten an umliegende Knoten
 - Erkennung von Paket-Dubletten
 - Erkennung von Loops im Netzwerk
 - Nur ein Active Messaging-Kanal
 - Interne Unterscheidung mehrerer Datenströme
 - Vermeidung der Interferenz mit anderen Knoten
 - Nach jedem erfolgreichen Paketversand wird für eine beschränkte, zufällige Wartezeit der Sendebetrieb unterbrochen

LTP ↔ CTP

- **Routing: Im Wesentlichen eine Kopie von CTP**
 - Anderer Active-Messaging-Kanal, da ansonsten keine von CTP abweichenden Basisstationen möglich
- **Daten: Größere Änderungen, aber Basis ebenfalls CTP**
 - Unterscheidung der Datenströme entfernt
→ Vereinfachung der Komponentenstruktur
 - Sequenznummern durch Zeitstempel ersetzt
 - Zeitstempel eignen sich auch als Sequenznummern und sind gleichzeitig für den Entwickler von großem Interesse
 - Mehrere Zeitstempel im Paket werden inkrementell als Zeitdifferenzen (Timediffs) abgelegt
→ Verringerte Datenmenge
 - Diverse effizienzsteigernde Maßnahmen sowie Fragmentierung eingeführt
 - Näheres auf den folgenden Folien!

LTP: Paketzusammenfassung

- **Ziel: Datenaufkommen senken, Pakete auslasten**
 - Header besitzen eine konstante Größe
 - Je mehr Nutzdaten ein Paket besitzt, desto geringer der Anteil der Header am Gesamtumfang
- **Herangehensweise:**
 - Mehrere Ereignismeldungen werden in einem gemeinsamen Paket übermittelt, sofern ihre Timediffs noch darstellbar sind
- **Verschiedene Zusammenfassungsverfahren analysiert**
 - Selbe/Verschiedene Absender und/oder
 - Gleiche/verschiedene Event Ids
- **Unterschiedliche Kosten:**
 - Rechenzeit auf Absender und weiterleitenden Knoten
 - Länge der LTP-Paketformate
- **Optimaler Ansatz: Abhängig vom Anwendungsfall**
- **Guter Kompromiss: Ein Paket kann Meldungen unterschiedlicher Art (Event Id) vom selben Absender umfassen**

LTP: Fragmentierung

- **Problem: Ein einzelnes Paket kann nur 17 Bytes Ereignisdaten aufnehmen**
 - Für größere Datenstrukturen, z.B. Routingtabellen, ist das zu wenig!
 - Eher seltener Anwendungsfall, soll die Header nicht zu weit vergrößern
- **Lösung: Paket-globale Fragment-Nummer**
 - Bezieht sich immer auf die letzte Ereignismeldung im Paket
 - Das letzte Fragment-Paket kann damit nur eine einzige Ereignismeldung enthalten!
 - Einschränkung der Paketzusammenfassung
 - Akzeptabel, da Sonderfall!
- **Derzeit noch nicht implementiert**

LTP: Auslastungsabschätzung

- Ziel: LTP soll den Betrieb drosseln oder einstellen, wenn andere Protokolle das Netzwerk benötigen
 - Randomisierte Wartezeiten wie in CTP nicht ausreichend
- Herangehensweise: FreeAirtimeEstimator (FAE)
 - TinyOS Active Messaging erweitert, meldet über Events
 - Empfang von Paketen
 - Status der Sendewarteschlange
 - FAE ermittelt anhand dieser Events die durchschnittliche Ruhezeit zwischen zwei Paketen im Netzwerk
 - Pakete, die LTP zuzurechnen sind, gehen nicht in Berechnung ein!
 - Ablauf:
 - Möchte LTP senden, stellt es eine Anfrage an den FAE
 - FAE wartet ein Drittel der aktuellen Ruhezeit, maximal 100ms
 - Falls in dieser Zeitspanne kein Netzwerkverkehr auftritt, darf LTP senden
 - Andernfalls Neustart des Wartevorgangs
 - FAE ist damit auf periodisch sendende Anwendungen zugeschnitten
 - Austausch der Planungsstrategie nach Bedarf möglich

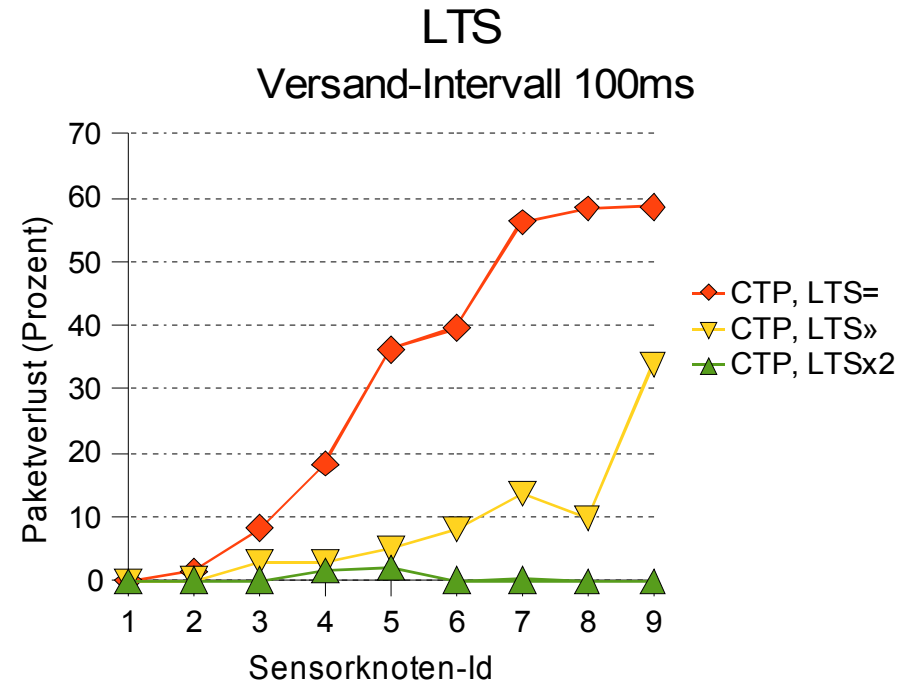
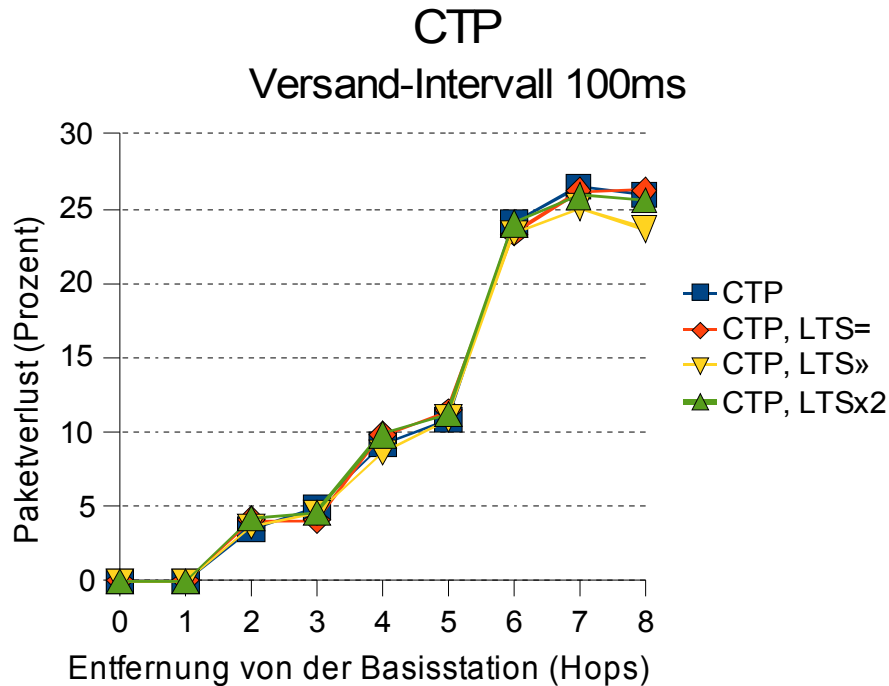
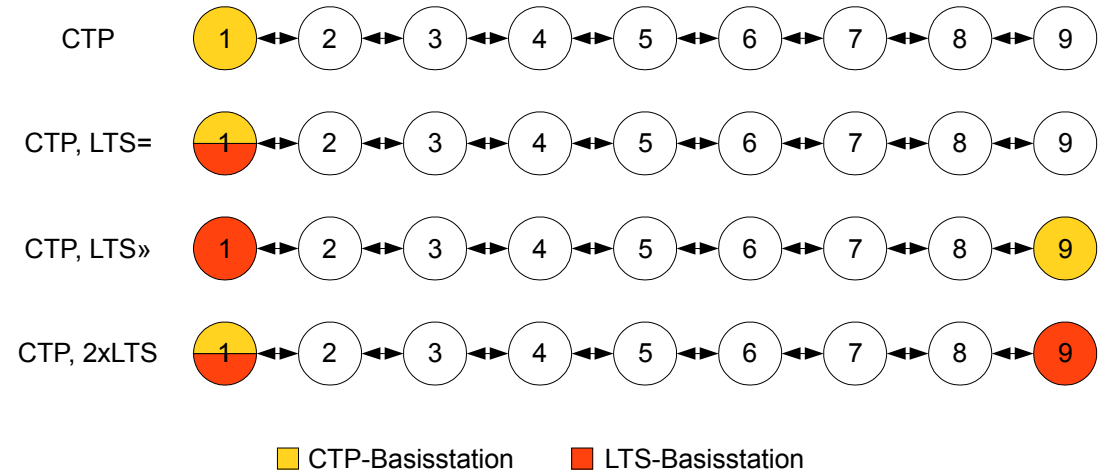
LTP: Priorisierung

- **Ziele:**
 - Bei Überlastung des Netzwerks wichtige Meldungen bevorzugt ausliefern
 - Paketverlust soll sich vorrangig auf unwichtige Meldungen auswirken
- **Herangehensweise:**
 - Forwarding Engine behandelt Pakete von hohem Gewicht bevorzugt
 - Markierung der Pakete mit dem maximalen Gewicht der enthaltenen Meldungen
 - Eine wichtige Meldung kann die Auslieferung mehrerer unwichtiger erzwingen
 - Akzeptabel aufgrund anderer Vorteile:
 - Kompakter Header
 - Vereinfachte Paketzusammenfassung auf dem Absender:
Kombinierbarkeit aller Gewichtsklassen in ein Paket → bessere Paketauslastung
- **Derzeit eingeschränkte Funktion**
 - Unwichtige Pakete werden nicht aus der Warteschlange verdrängt
 - Kein Eingang neuer wichtigerer Pakete aufgrund der Congestion-Nachrichten
 - Nur auf eigenem Knoten erzeugte Pakete könnten für Verdrängung sorgen
 - Congestion-System muss neu überdacht werden

LTP: Evaluation (vs. CTP)

Messung von Paketverlusten

- Vier Varianten für die Lage der Basisstationen
- Weitere Intervalle evaluiert
 - 40ms: Leicht erhöht
 - 400ms: Verluste < 0,5%



LTS nesC-Compiler

- **Aufgaben des angepassten Compilers:**
 - Schreiben des Sensorknoten-Deskriptors
 - Umsetzung der Annotationen/expliciten Protokollierungen in nesC-Code
- **Basis: nesc (offizieller nesC-Compiler)**
 - Schreibt C-Datei
 - Diese wird im Anschluss von einem plattformspezifischen GNU C-Compiler (gcc) in Maschinencode überführt
 - Basiert selbst auf gcc 2.8.1
 - Ältere C-Grammatik
 - Phasen
 - Lexikalische Analyse (Lexer)
 - Syntaxanalyse (Parser): Auf Basis von yacc/bison
 - Kontextfreie Grammatik definiert Regeln
 - Während der Abarbeitung der Regeln werden Aktionen durchgeführt
 - nesc erzeugt abstrakten Syntaxbaum (AST) aus verzeigerten C-Structs

Schreiben des Sensorknoten-Deskriptors

- Während des Parsings stehen bereits alle erforderlichen Informationen bereit
- Alle nötigen Daten sind im AST zu finden
- Insbesondere: Typinformationen in type-Structs
 - Hierarchisch verkettet
 - type-Struct für Array zeigt auf type-Struct des Element-Typs
 - type-Struct für Struct zeigt auf type-Structs der Komponenten-Typen
 - type-Struct für Pointer zeigt auf type-Struct des dereferenzierten Typs
 - Konzept sehr ähnlich zu XML-Dateiformat der Sensorknoten-Deskriptoren
- nescc enthält bereits grundlegende XML-Funktionen
 - Arbeiten auf XML-Tag-Ebene
 - Werden nun auch zur Erzeugung des Sensorknoten-Deskriptors eingesetzt

Erweiterung von Modulen (1)

- Für Logging ist zusätzlicher Code erforderlich
 - Beispiel: Einfache Annotation:

```
#include <Lts.h>

module SimpleC {

    .....
}
implementation {
    void simpleFunc(uint8 t param)
        @logInvocation(SEVERITY_ERROR, "%d", "param") {

        uint16_t localVar = 0;
        .....

    }
    .....
}
```

Erweiterung von Modulen (1)

- Für Logging ist zusätzlicher Code erforderlich
 - Beispiel: Einfache Annotation:

```
#include <Lts.h>

module SimpleC {
  uses interface Logging;
  .....
}

implementation {
  void simpleFunc(uint8_t param)
    @logInvocation(SEVERITY_ERROR, "%d", "param") {
    call Logging.start(SEVERITY_ERROR, 42, sizeof(param));
    call Logging.appendData(&param, sizeof(param));
    call Logging.finish();

    uint16_t localVar = 0;
    .....

  }
  .....
}
```

- Interface einbinden
- Parameterbezeichner auswerten
- Command-Calls für Logging-System einfügen

Erweiterung von Modulen (1)

- Für Logging ist zusätzlicher Code erforderlich
 - Beispiel: Einfache Annotation:

```
#include <Lts.h>

module SimpleC {
  uses interface Logging;
  .....
}

implementation {
  void simpleFunc(uint8_t param)
    @logInvocation(SEVERITY_ERROR, "%d", "param") {
    call Logging.start(SEVERITY_ERROR, 42, sizeof(param));
    call Logging.appendData(&param, sizeof(param));
    call Logging.finish();
    {
      uint16_t localVar = 0;
      .....
    }
  }
  .....
}
```

- Interface einbinden
- Parameterbezeichner auswerten
- Command-Calls für Logging-System einfügen
- nesC-Grammatik berücksichtigen (Variablendeklarationen)

Erweiterung von Modulen (2)

- Manuelle Erweiterung des ASTs wird mit steigender Größe des einzufügenden Codes schnell komplex und unübersichtlich
 - Lösung zur Modul-Erweiterung: Vorhandenen Lexer/Parser nutzen
- **Code Injection-System**
 - Lexer in nesc verwendet zum Lesen von Code zwei Funktionen:
 - Lesen eines einzelnen Zeichens aus der aktuellen Eingabedatei
 - Zurückstellen eines einzelnen Zeichens in einen Puffer der C-Bibliothek
 - Beide Funktionen neu definiert
 - Eigener, großer Puffer für Rückstellung
 - Code Injection befüllt Puffer mit dem einzufügenden nesC-Code
 - Erzeugung des einzufügenden nesC-Codes und Aufruf der Code Injection:
Durch Erweiterungen der Regel-Operationen des Parsers
 - Interface-Einbindung: Wird in jedem Modul eingefügt
 - nesc berücksichtigt das Interface aber grundsätzlich nur bei Bedarf!
 - Command-Calls: Einfügung nur falls erforderlich und abhängig von Annotation / expliziter Protokollierung

Erweiterung von Configurations (1)

- Für Logging sind zusätzliche Wirings erforderlich
 - Das von protokollierenden Modulen verwendete Interface „Logging“ muss von einer anderen Komponente bereitgestellt werden

```
configuration WiringC {  
}  
implementation {  
  components ModuleWithLoggingP as MWL;  
  components DefaultLoggingC;  
  
  MWL.Logging -> DefaultLoggingC.Logging;  
  .....  
}
```

- Logging-Komponente referenzieren
 - DefaultLoggingC oder RootLoggingC, abhängig von der Rolle des Sensorknotens
- Verbindung herstellen
- Erweiterung von Configurations erfolgt direkt im AST
 - Geringer Aufwand, da nur wenige kompakte AST-Knoten zu erzeugen

Erweiterung von Configurations (2)

- Welche Module benötigen ein Wiring?
- **Verarbeitungsprozess:**
 - Parser verarbeitet eine Configuration
 - Parser verarbeitet die darin referenzierten Module
 - Enthält ein Modul LTS-Anweisungen: Füge den Namen des Moduls einer Warteliste hinzu
 - Nach Abschluss der Verarbeitung der Configuration:
 - Betrachte die in dieser Configuration referenzierten Module
 - Falls der Name eines Moduls in der Warteliste enthalten ist:
 - Falls das Modul generisch ist, füge ein Wiring hinzu
 - Instanzen generischer Module sind nur aus der erzeugenden Configuration sichtbar, das Wiring kann nur an dieser Stelle hergestellt werden
 - Da jede Instanz eine isolierte Komponente darstellt, benötigt sie auch ihr eigenes Wiring
 - Falls das Modul nicht generisch ist, füge nur dann ein Wiring hinzu, wenn noch in keiner anderen Configuration ein Wiring für dieses Modul angelegt wurde
 - Information über die Existenz eines Wirings ist ebenfalls Teil der Warteliste

Zusammenfassung

- **Logging Tree System: Ein generisches, verteiltes, effizientes Logging- und Tracing-Framework für TinyOS**
 - Betrieb über das Netzwerk mit nur wenigen verkabelten Basisstationen
 - Einfache Anwendung durch den Benutzer
 - Weitreichende Logging- und Tracing-Möglichkeiten inkl. beliebiger Texte sowie komplexer Datenstrukturen
 - Wichtigste Bestandteile des Systems:
 - Angepasster nesC-Compiler
 - Komfort: Umsetzung einfacher Logging-Anweisungen in komplexen Code
 - Effizienz: Extraktion statischer Teile von Logging-Meldungen, die im laufenden Betrieb nicht übermittelt werden müssen
 - ConvergeCast-Netzwerkprotokoll, speziell auf Logging abgestimmt
 - Paketzusammenfassung
 - Auslastungsabschätzung
 - Priorisierung
 - Fragmentierung

- Weiterentwicklung des bestehenden Systems
 - Evaluation weiterer Verfahren zur Paketzusammenfassung
 - Verbesserung der Priorisierung, Umsetzung der Fragmentierung
 - Unterstützung für bedingtes Tracing
- Grundlegend neue Funktionen oder Systeme
 - Konfigurierbarkeit des Logging-Frameworks im laufenden Betrieb
 - Beispielhafte Anwendungsfälle:
 - Hinzufügen und Löschen von Loggings/Tracings
 - Aktivieren/Deaktivieren von Loggings/Tracings
 - Erfordert bidirektionales Routing- oder Flooding-Protokoll
 - Reduzierung der Änderungen am nesC-Compiler mit Hilfe eines aspektorientierten Systems
 - Einweben von Code sollte sich auf diesem Wege erreichen lassen
 - Erzeugung des Sensorknoten-Deskriptors geht jedoch über den Funktionsumfang eines üblichen aspektorientierten Systems hinaus

Fragen?

