Experiences from a Decade of TinyOS Development

Presented by Surya Bakshi

TinyOS

- Developed in UC Berkley
- Operating system for sensor network embedded devices
- Specialized C dialect, nesC, with compiler

Why?

- Sensor networks exist in many commercial industries
  - Smart meters, industrial monitoring systems

- TinyOS has emerged as the first widely adopted microcontroller operating system

- Two major successes

Importance of TinyOS

- First reason is it’s success leaving academia and impacting industry work
  - But remained a heavily researched system despite widespread adoption
- Second it provided a novel programming language, nesC
- nesC, allows new programming models to be adopted
  - Re-using operating system design and language constrains possibilities

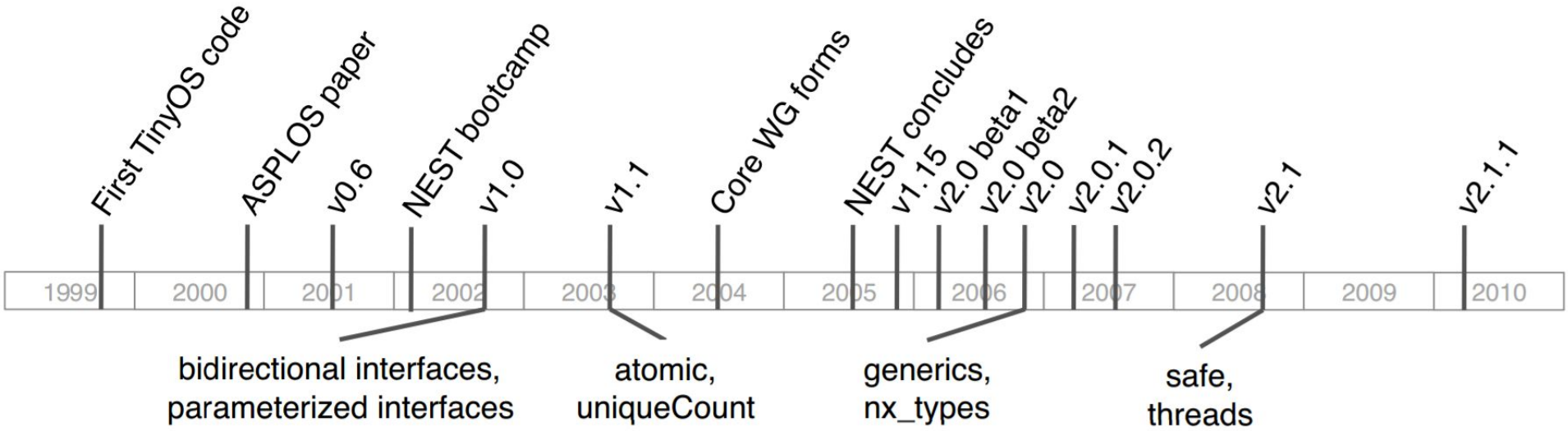

Figure 1: Timeline of major events in TinyOS development from 1999-2010.

Goals

- Examine the design decisions of TinyOS
- Evaluate their benefits/costs
- Draw lessons learned from design decisions
- How could it have been better?

TinyOS Design Goals

Minimization:

- Energy: modern computer architectures just don't cut it
- Device is asleep most of the time
- **1.3-2 μA vs 950 μA - sleep current**

For 100,000 units $6 cost cut yields **$600,000** savings

TinyOS Design Goals

<u>Minimization</u>:

- Energy: modern computer architectures just don't cut it
- Device is asleep most of the time
- **1.3-2 *μ*A vs 950 *μ*A - sleep current**

<u>Bug Prevention</u>:

- Minimization means a lack of debugging tools
  - Debug logs, lots of devices
- User frustration with debugging
- Patches needed for *all* devices

RAM and ROM

- ROM minimization performed by nesC compiler (inlining, dead code elimination)

- RAM minimization done mostly through software structures

* inlining: replace function call with function code
* dead code elimination: remove code that does not impact result

Example: ADC

- Need to manage configuring ADC for use by a driver
- **Intuitive approach**: each driver has configuration structure it passes in
- *What were the outcomes?*

Example: ADC

- Need to manage configuring ADC for use by a driver

- **Intuitive approach**: each driver has configuration structure it passes in

- Each driver had to take up space in RAM for its own structure even though only 1 is needed at any time
- **Design**: generate structure at runtime on the stack

Example: ADC

- Need to manage configuring ADC for use by a driver
- **Intuitive approach**: each driver has configuration structure it passes in
- Each driver had to take up space in RAM for its own structure even though only 1 is needed at any time
- **Design**: generate structure at runtime on the stack

Benefits: save 4 bytes RAM / client
Cost: ROM use up by 50-60 bytes

Example: Timer

- An average timer can require anywhere from 3-15 timers (10 bytes of state per)

- Timer maintained linked list of timer states
  - 20% overhead in memory usage
  - Dynamic memory allocation introduced a lot of bugs

Example: Timer

- An average timer can require anywhere from 3-15 timers (10 bytes of state per)

- Timer maintained linked list of timer states
  - 20% overhead in memory usage
  - Dynamic memory allocation introduced a lot of bugs

- Unique
  - Counts number of calls with string s
  - But RAM usage still increased

Example: Timer

- An average timer can require anywhere from 3-15 timers (10 bytes of state per)

- Timer maintained linked list of timer states
  - 20% overhead in memory usage
  - Dynamic memory allocation introduced a lot of bugs

- Unique
  - Counts number of calls with string s
  - But RAM usage still increased (structure array)
- uniqueCount
  - `timer_state_t timers[uniqueCount("Timer")];`

Isolation

- Initially, no dynamic memory allocation

Problems:

- Fixed size scheduling array - *how to wait?*
- Packet transmission queues - *any bad app breaks everything*

How to reconcile?

Isolation

Problems:

- Fixed size scheduling array - *how to wait?*
- Packet transmission queues - *any bad app breaks everything*

Design: static virtualization

- Give every application a virtual instance of a resource
- Let compiler allocate appropriate memory, **not the user!**
- Each app's api state affected by **no one else**

Tradeoff: Increased ROM usage → Fewer crippling bugs

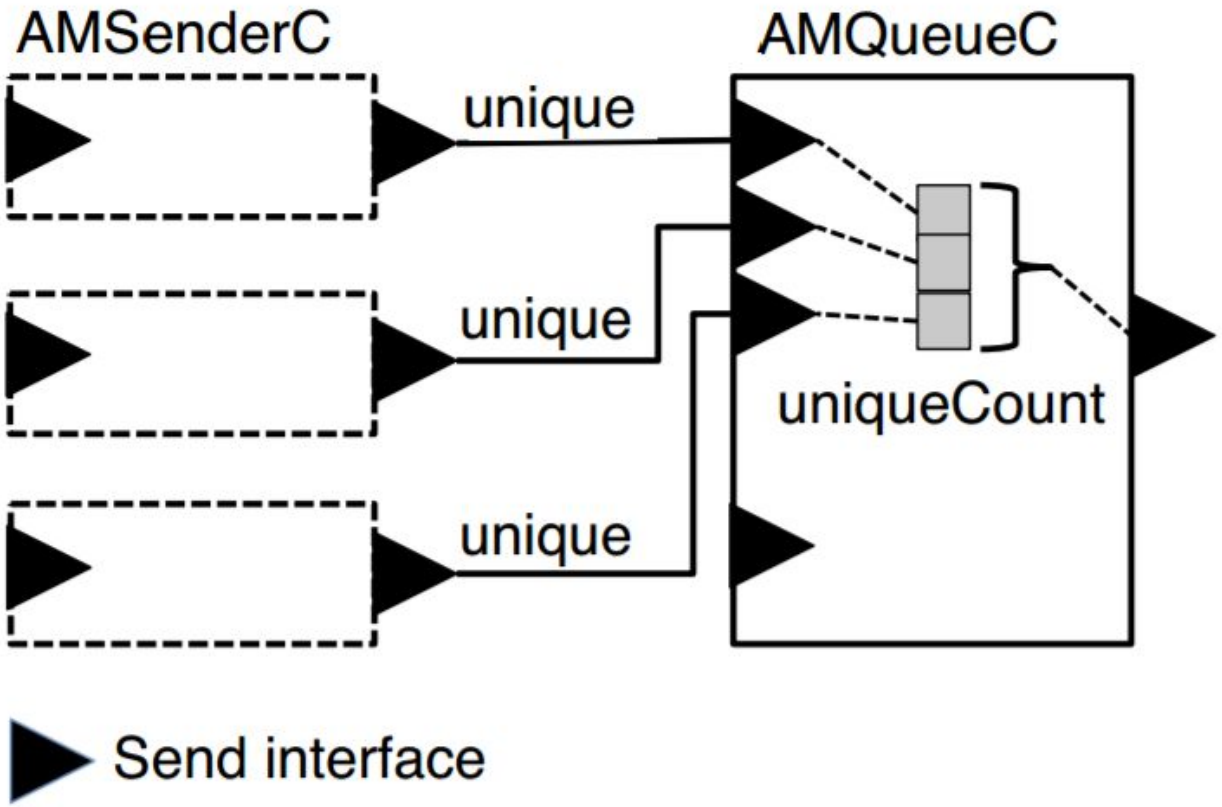

Figure 2: Static virtualization with AMSenderC.

Language

- nesC co-developed with tinyOS
- A new language and new operating system opens the door to new programming models
  - Static virtualization
- nesC has continued to move further away from C

Language

- nesC co-developed with tinyOS
- A new language and new operating system opens the door to new programming models
  - Static virtualization
- nesC has continued to move further away from C

Tradeoff (use): highly flexible in its domain → barrier to entry continues to grow

Language

- nesC co-developed with tinyOS
- A new language and new operating system opens the door to new programming models
  - Static virtualization
- nesC has continued to move further away from C

Tradeoff (use): highly flexible in its domain → barrier to entry continues to grow
Tradeoff (adoption): commercial and academic use cases → barrier to simple diy/hobby projects

Language

- nesC co-developed with tinyOS
- A new language and new operating system opens the door to new programming models
  - Static virtualization
- nesC has continued to move further away from C

Tradeoff (use): highly flexible in its domain → barrier to entry continues to grow
Tradeoff (adoption): commercial and academic use cases → barrier to simple diy/hobby projects
Tradeoff (competition): Contiki now used more than tinyOS in research

Components

Separate **interface and implementation**, provide data **privacy** and allow code **reuse.**

- Intended as a research tool
- Abstracting makes modifying and experimenting easier

Tradeoff: easy to experiment → very large barrier for entry into TinyOS

Examples

Radio Driver:

- ~2400 lines of code abstracted into 40 files

Timer:

- Code to convert counter to millisecond counter less than 1kB of code
- Spread out across 8 different files

Community

- Started off as a research project for students at **UC Berkley**
- Progress through first two iterations was done as collaboration with Berkley and **TU Berlin**
- Industrial involvement critical in development
- However commercial goals differ greatly from academic ones

Community - Industry

- Industry development was vital in adding critical features to TinyOS
  - Moteiv and ArchRock start up

Community - Industry

- Industry development was vital in adding critical features to TinyOS
  - Moteiv and ArchRock start up
- But, industry and academic goals are very different
  - **“Good enough”** is good enough for a company wanting to start monetization

Community - Industry

- Industry development was vital in adding critical features to TinyOS
  - Moteiv and ArchRock start up
- But, industry and academic goals are very different
  - **“Good enough”** is good enough for a company wanting to start monetization
- Ultimately ArchRock and Moteiv both forked private repos from TinyOS
  - Conflicts arose with versions of TinyOS
  - Moteiv staff also work with academic development

Community - Users

- The rapid use and download of TinyOS forced developers to support users
- Many questions were about **“simple”** things that could be inferred with work
- As a result, the user base inspired effort to create **verbose documentation**
- Further advancing TinyOS adoption in **both** academic and commercial pursuits

Tradeoff: A lot of time spent doing this → Support became a lot easier

Lessons Learned from TinyOS Development

Good: Language Extensions

- nesC was needed to allow robust code that minimizes hardware
- Using C would have hindered TinyOS popularity

Lessons Learned from TinyOS Development

Good: Language Extensions

- nesC was needed to allow robust code that minimizes hardware
- Using C would have hindered TinyOS popularity

Bad: How They Evolved

- Making it easier to solved hard problems can make it hard to solve simple ones

Lessons Learned from TinyOS Development

Good: Software Components

- Cleaner interfaces, more robust code
- Reusable code, flexibility in building

Lessons Learned from TinyOS Development

Good: Software Components

- Cleaner interfaces, more robust code
- Reusable code, flexibility in building

Bad: Component Architecture

- In practice, clean, easy-to-use code without too much structural complexity can be easier to copy/modify

Lessons Learned from TinyOS Development

Good: Initial Users

- Encourage research use internally
- Get funding for development

Lessons Learned from TinyOS Development

Good: Initial Users

- Encourage research use internally
- Get funding for development

Bad: Focusing on Experts

- Should have focused on broad participation
- Too complicated for DIY and hobby users

Lessons Learned from TinyOS Development

Good: Late Industrial Involvement

- Industry partners contributed a lot of code to TinyOS
- Many drivers that make up widely used code today

Lessons Learned from TinyOS Development

Good: Late Industrial Involvement

- Industry partners contributed a lot of code to TinyOS
- Many drivers that make up widely used code today

Bad: Early Industrial Involvement

- Different goals of correctness vs pace of development

Lessons Learned from TinyOS Development

Good: Diverse Documentation

- Save support time with only good questions
- Saves time adding new people

Lessons Learned from TinyOS Development

Good: Diverse Documentation

- Save support time with only good questions
- Saves time adding new people

Bad: Only Developer Doc

- Aim for documentation that reaches many levels of technical background

Questions?

Discussion

1. There is a clear barrier to entry for understanding the system, but is it really that bad to sacrifice user work for better/cleaner/reusable code?

Discussion

1. There is a clear barrier to entry for understanding the system, but is it really that bad to sacrifice user work for better/cleaner/reusable code?

2. Would going with “good enough” have helped TinyOS adoptions, usage or hindered it? Consider all the features that were carefully added and tightened for minimization.

Discussion

1. There is a clear barrier to entry for understanding the system, but is it really that bad to sacrifice user work for better/cleaner/reusable code?

2. Would going with “good enough” have helped TinyOS adoptions, usage or hindered it? Consider all the features that were carefully added and tightened for minimization.

Scribe Report

Criticisms

- This paper does not appear to be research as we are generally used to seeing it. Seems to be more like a workshop paper rather than a conference paper. It mostly recounts already known facts and gives some comments on it.

- Doesn’t offer opinions of others who evaluate TinyOS.

Discussion

1. Should the paper have focused on less on personal opinion and more on opinions from academia and industry? Would that give more legitimacy to the paper?

2. You guys are working on projects now. Have you created any documentation yet? When is the best time to create support structure for other users?

3. This paper got criticism for not being “research”. Is there merit to this paper as research? Is this an important topic of discussion? Does someone have a conflicting opinion on the merit of this kind of work?