Towards Dynamically Composed Real-Time Systems

Leandro Batista Ribeiro
lbatistaribeiro@tugraz.at

16 November 2017

Institute of Technical Informatics
Embedded Automotive Systems Group
Graz University of Technology

Agenda

- Introduction
- Software Generation
- Update Mechanisms
- Real-Time Awareness

Agenda

- Introduction
- Software Generation
- Update Mechanisms
- Real-Time Awareness

Real-Time Systems

Real-time systems operate on time constraints.
Reaction to events or inputs must occur within a defined time window – not too late, not too soon.

Failure on keeping the time requirements might result in:

- Heavy damages in **hard real-time** systems
- Undesirable, but tolerable problems in **soft real-time** systems

Dynamically Composed Systems

- Modular systems
- Partial updates
- On-the-fly updates

Dynamically Composed Real-Time Systems

- Modular systems
- Partial updates
- On-the-fly updates

- System remains real-time at any point in time: before, during and after any upadtes.

Dynamically Composed Real-Time Systems

- Modular systems
- Partial updates
- On-the-fly updates

- System remains real-time at any point in time: before, during and after any upadtes.

Multi-Core Sustainable modular adept real-time Operating System

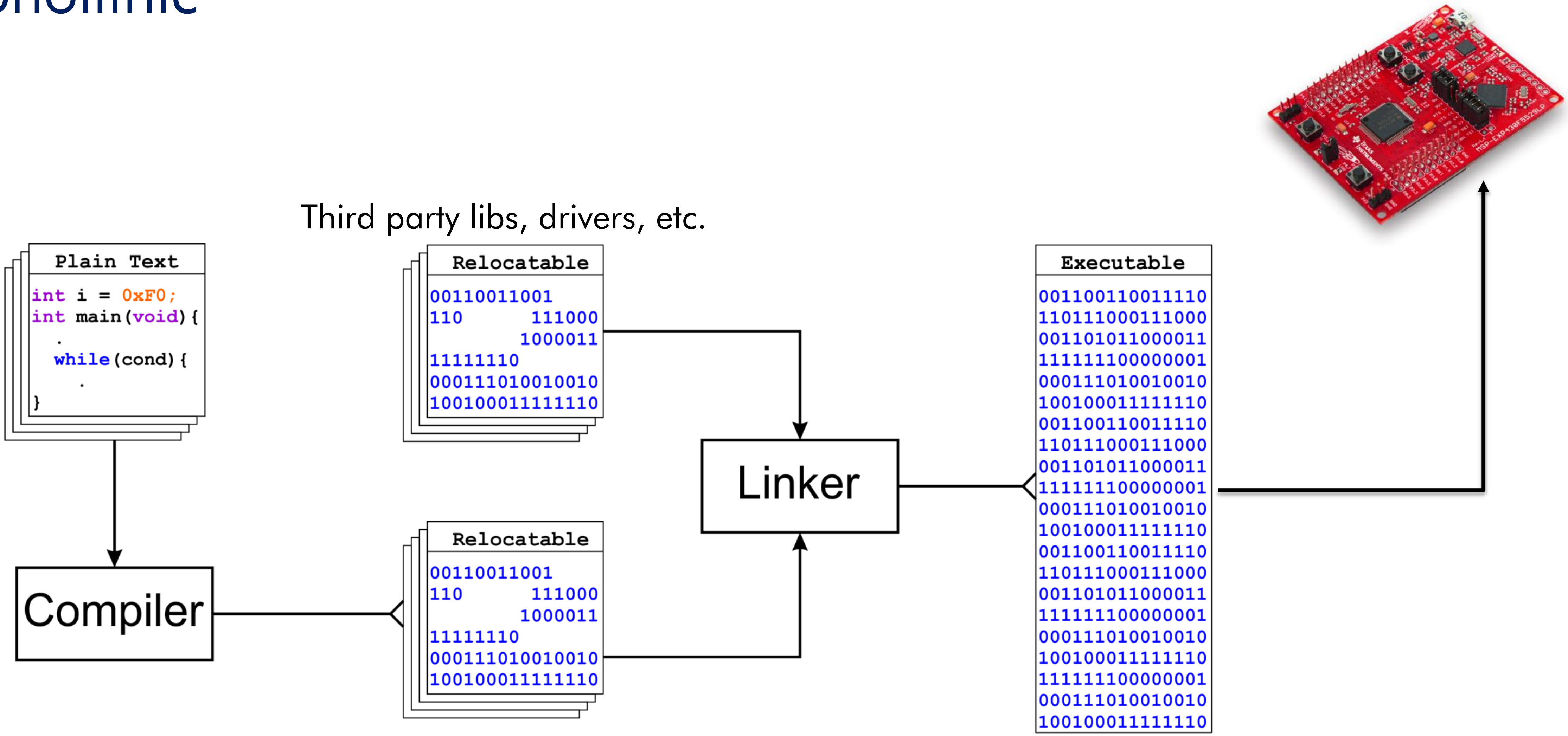
Motivation

- Internet of Things (IoT) → Billions of devices
- More software customization → Software diversity
- More software providers → Access restrictions
- More classes of systems → Common processor and services

Agenda

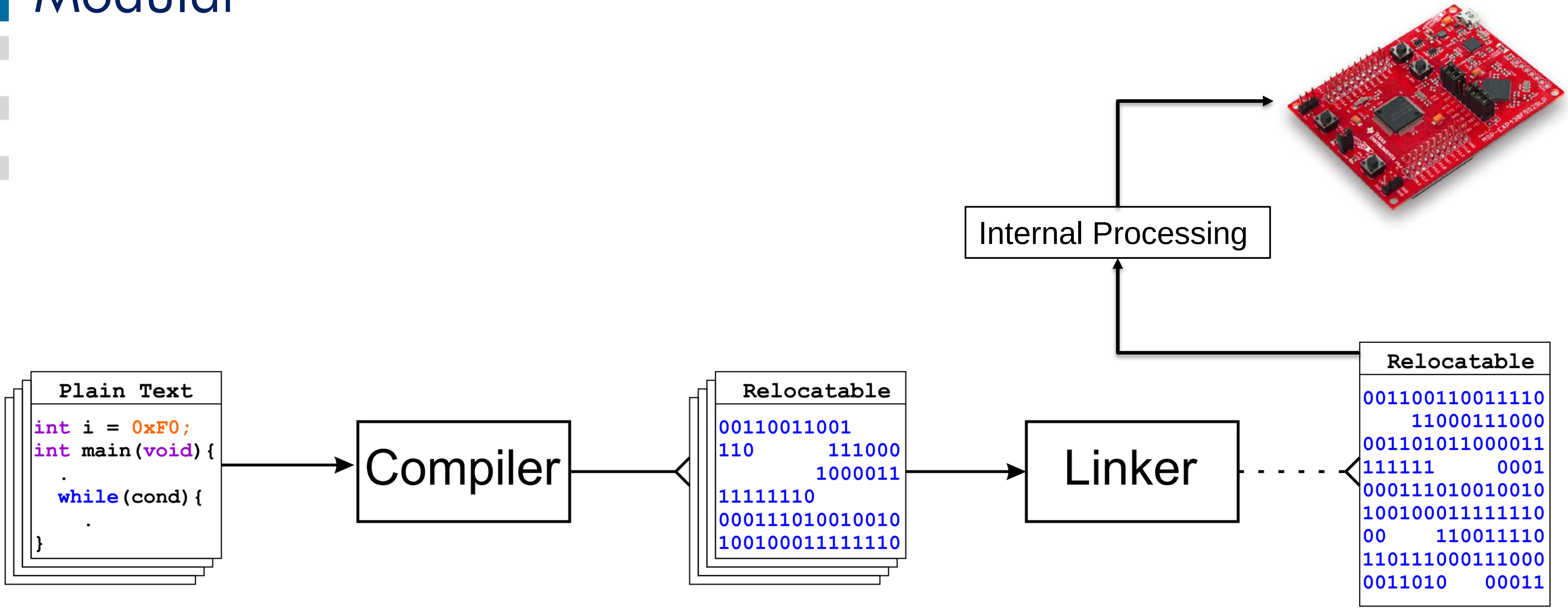
- Introduction
- Software Generation
- Update Mechanisms
- Real-Time Awareness

Monolithic



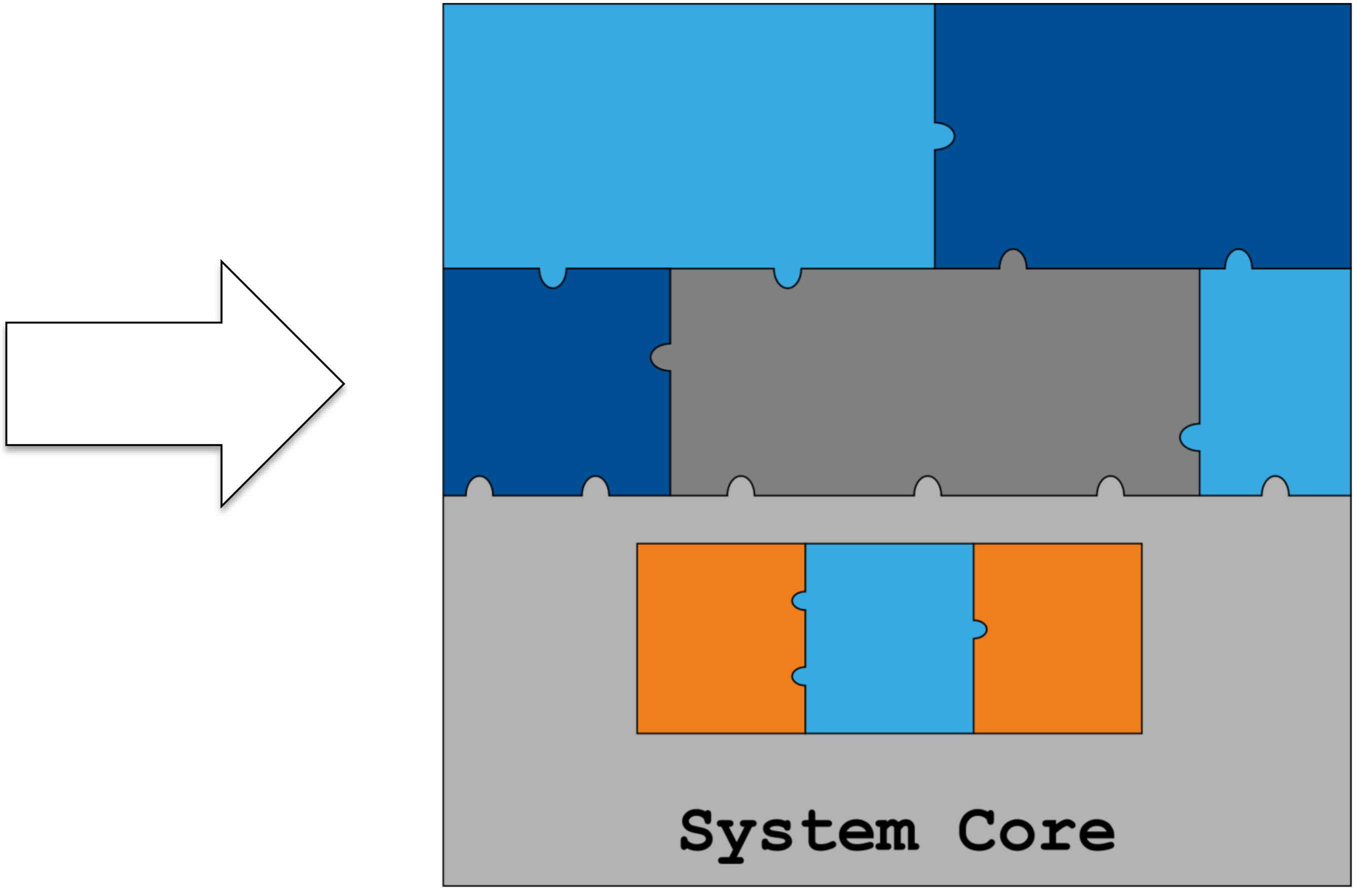
Third party libs, drivers, etc.
Plain Text
int i = 0xF0;
int main(void){
while(cond){
}
Compiler
Relocatable
00110011001
110 111000
1000011
11111110
000111010010010
100100011111110
Relocatable
00110011001
110 111000
1000011
11111110
000111010010010
100100011111110
Linker
Executable
001100110011110
110111000111000
001101011000011
111111100000001
000111010010010
100100011111110
001100110011110
110111000111000
001101011000011
111111100000001
000111010010010
100100011111110
001100110011110
110111000111000
001101011000011
111111100000001
000111010010010
100100011111110
111111100000001
000111010010010
100100011111110

Modular



Plain Text
int i = 0xF0;
int main(void){
.
while(cond){
.
}
Compiler
Relocatable
00110011001
110 111000
1000011
11111110
000111010010010
100100011111110
Linker
Relocatable
001100110011110
11000111000
001101011000011
111111 0001
000111010010010
100100011111110
00 110011110
110111000111000
0011010 00011
Internal Processing

Modular Executable



Agenda

- Introduction
- Software Generation
- Update Mechanisms
- Real-Time Awareness

Why update?

- Bugfixes
- Security breaches
- New requirements/legislation
- Enhancements
- Reconfigurable hardware

How to update?

- Physical access

Normal Operation Disrupted?

- Remote updates
  - User awareness
  - Background updates

Our Focus

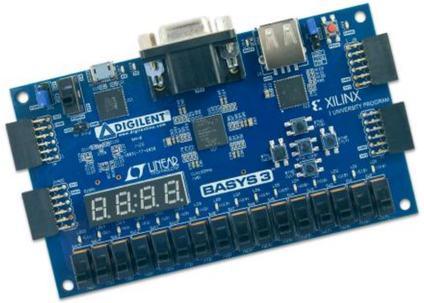
- Physical access

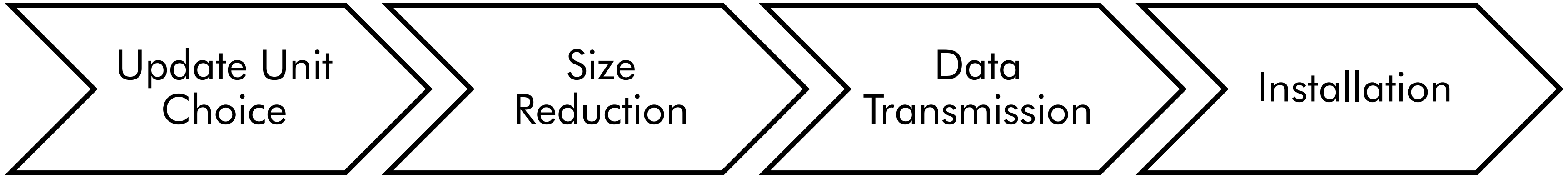
Normal Operation Disrupted?

No!

- Remote updates
  - User awareness
  - Background updates

General Steps





- Monolithic
- Native Module
- VM Module

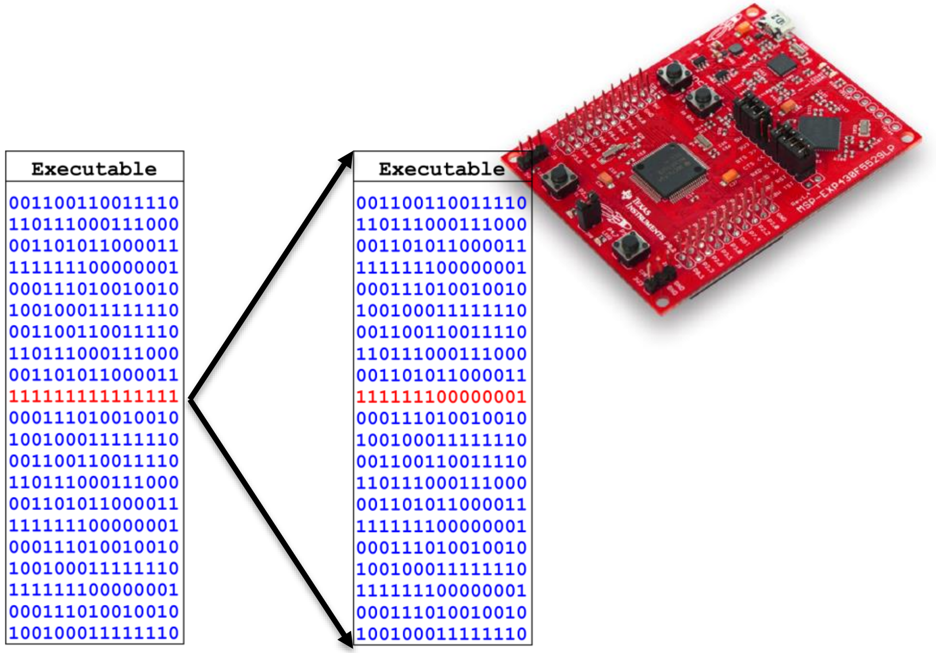
- ELF Optimizations
- Delta Files
- Compression

- Server-Client
- Dissemination Protocols

- Authenticity
- System Configuration
- Linking/Relocation
- Load and Execution

Update Unit – Monolithic

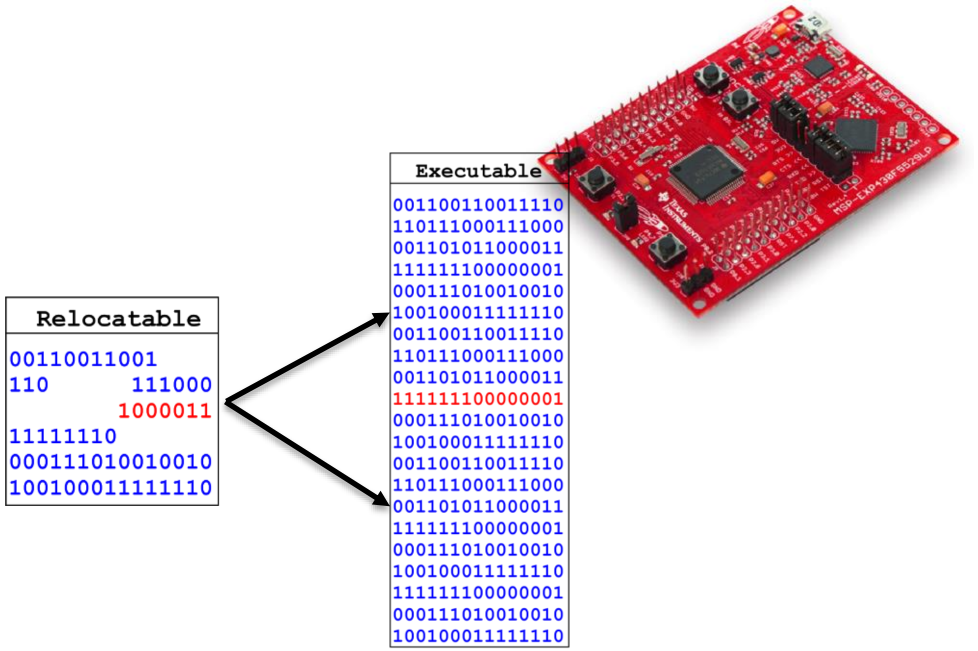
- Example: TinyOS



Update Unit – Native Modules

- Relocatable Code Only
  - Example: Contiki
  - 45%-55% metadata overhead [1]
  - ~13% faster than PIC [2]

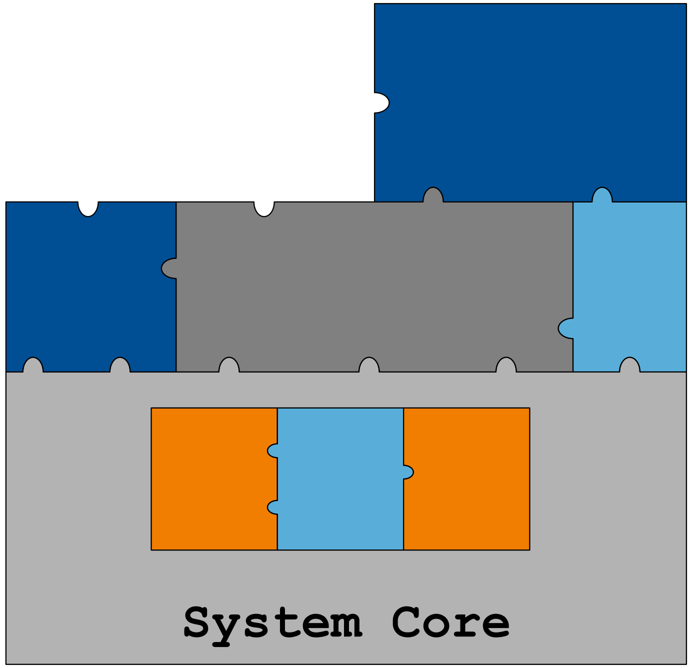
- Position Independent Code (PIC)
  - Example: SOS
  - Less metadata overhead
  - Compiler and architecture dependent



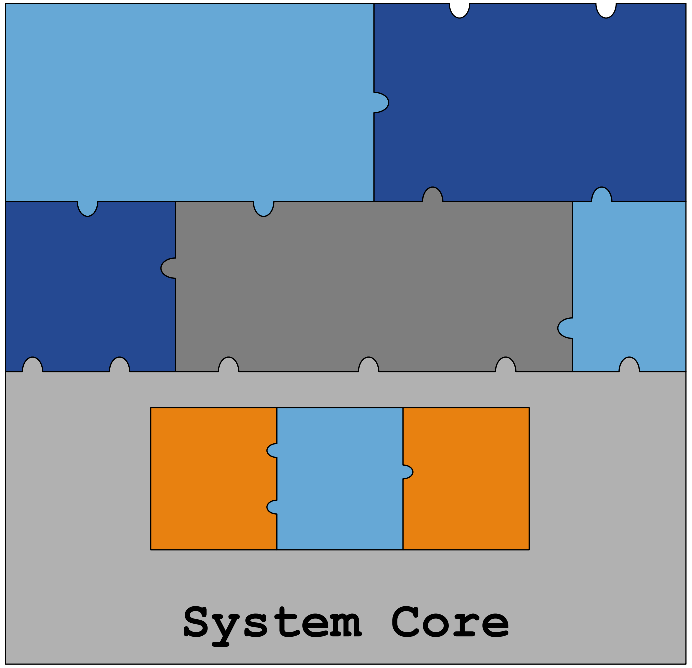
Update Unit – Native Modules

Potential Problems

- Removal of a module needed by other modules



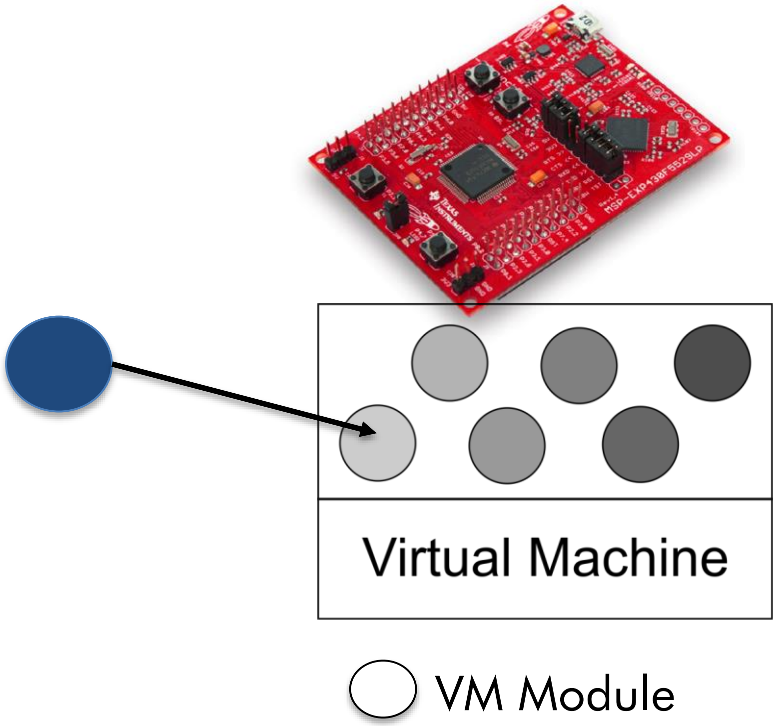
- Dependencies not present in current system



Update Unit – VM Module

VM execution overhead

- processing overhead due to code interpretation at runtime mostly outweighs the costs saved in the transmission [3]



Size Reduction

ELF Optimization

- CELF [3]
  - Fields size reduction 32/64 bits → 8/16 bits
  - CELF → typically ~ 50% of ELF

- SELF [4]
  - Fields size reduction
  - Tailoring of relocation, string and symbol tables
  - SELF → 15%-30% of ELF | 38%-83% of CELF
  - Loading speed 40%-50% of standard mechanism

Size Reduction

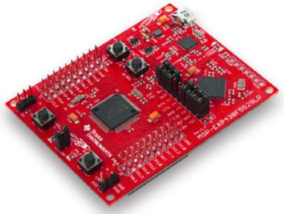
Delta Files

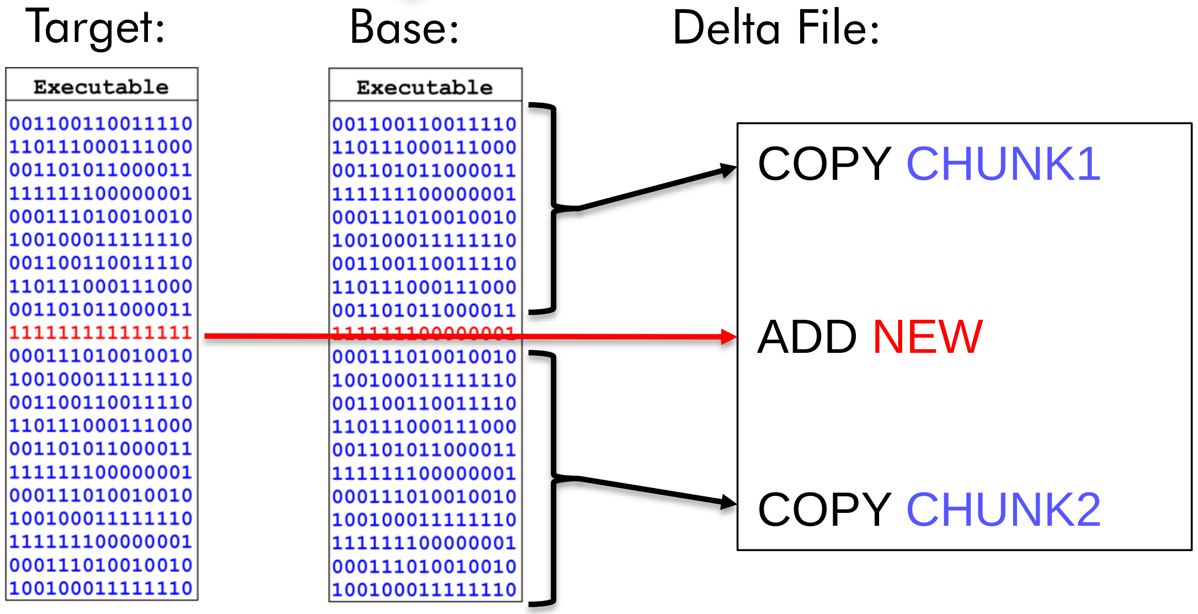
Incremental Approach

- Target version built on server
- Delta file generated on server
- Delta file transmitted
- Target version rebuilt on client

Techniques

- Slop regions[5]
- Similarity[6]





Size Reduction

Compression

- Decompression on client → More processing overhead
- Gzip on sensor nodes [7]

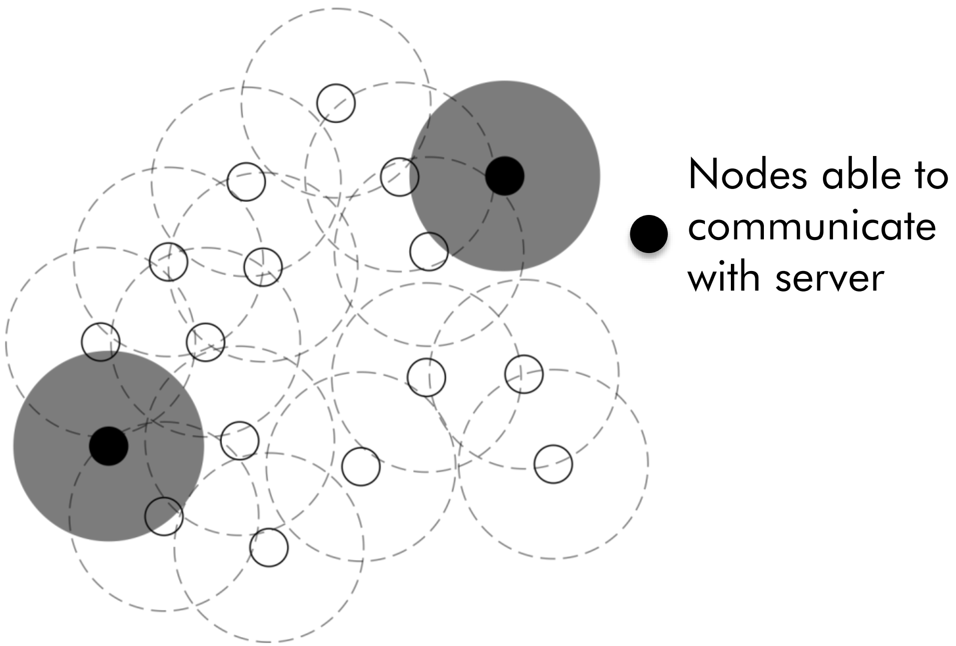
Data Transmission

Client - Server

- Point-to-point connection between server and target system

Dissemination Protocols

- Direct connection with some nodes
- Data distributed among remaining nodes



Installation

Authenticity Check

- Make sure updates are legit

System Configuration

- Check/resolve dependencies
- Set up control blocks (tasks, resources, etc)

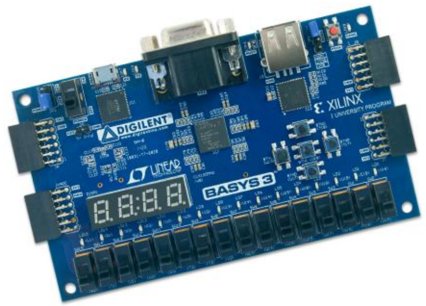
Linking/Relocation

- Transform a relocatable file in executable

Load/Execution

- Make software ready to run

Our Approach



Update Unit Choice → Size Reduction → Data Transmission → Installation

Update Unit Choice
- Monolithic
- Native Module
- VM Module

No PIC!

Size Reduction
- ELF Optimizations
- Delta Files
- Compression

Data Transmission
- Server-Client
- Dissemination Protocols

Installation
- Authenticity
- System Configuration
- Linking/Relocation
- Load and Execution

+

Real-Time Awareness

Agenda

- Introduction
- Software Generation
- Update Mechanisms
- Real-Time Awareness

Current Approaches

Simple scenarios

- Rate-monotonic scheduling
- No resource sharing
- No task synchronization

Examples:

- *A model for updating real-time applications* [8]
  - New WCET $\leq$ Old WCET
  - New modules stored in the heap
- *A method for dynamic software updating in real-time systems* [9]
  - Schedulability analysis before accepting update
  - Update finishes within two hyper-periods

Our Goals

Offer partial on-the-fly updates and make sure the system remains real-time at any point in time: before, during and after any modification.

- Unintrusive updates
- Runtime schedulability analysis
- Minimize execution/memory overheads
- Loose coupling
- Portability / Use of standards
- Support wide range of devices

Trade-offs

Efficient analysis and low memory overhead

- Too little metadata → More modules, slow or impossible analysis
- Too much metadata → Less modules, faster or easier analysis

Low execution overhead and loose coupling

- Few Indirections → Low execution overhead, modules strongly coupled
- Many indirections → High execution overhead, modules loosely coupled

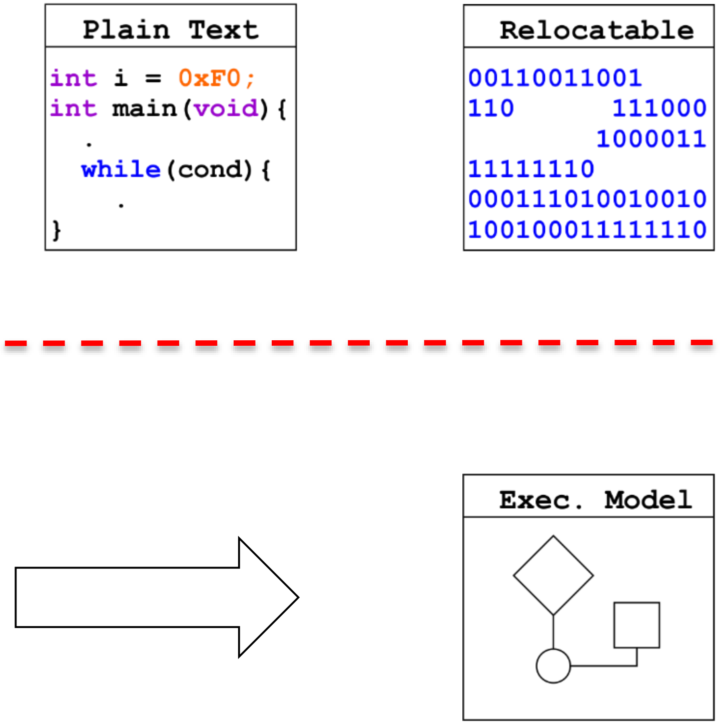
Generic updates and low client processing/memory overhead

- Server-only processing → Tailored updates, client simply loads the update
- Client-only processing → Generic updates, client analyzes and tailors the update

Our Approach

Metadata Analysis

- Memory layout and and size
- Symbols
- Version information
- Tasks configuration
- Modules dependencies
- Synchronization points
- Worst case execution time
- Worst case response time
- Interference time
- Priority inversions

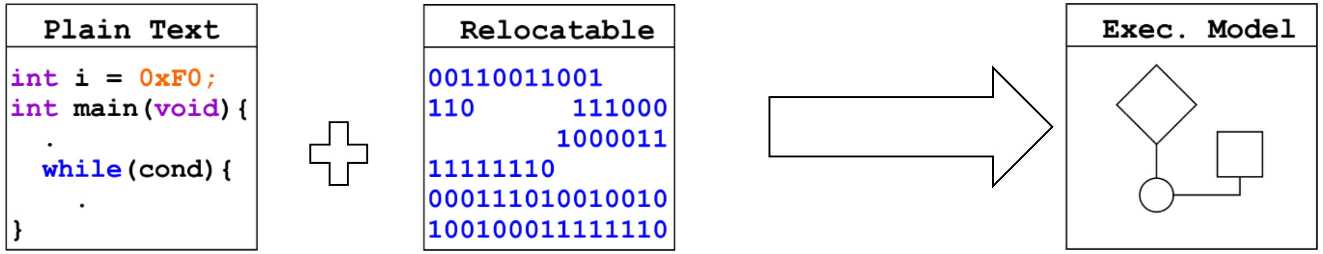


Our Approach

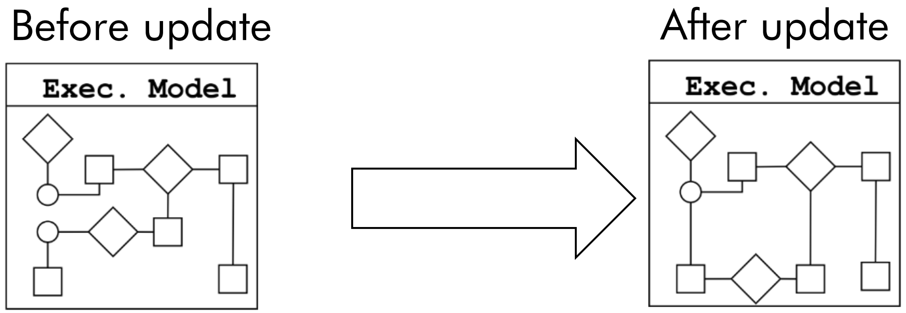
Execution Model

- Describe software execution

  - Individual modules
    - Tasks WCET
    - Synchronization pairs



  - Whole System
    - Tasks WCRT
    - Potential deadlocks or starvations



Our Approach

Updates will only be accepted if they are compatible with the system.

Compatibility

- Pluggability: Dependencies
- Interoperability: Execution

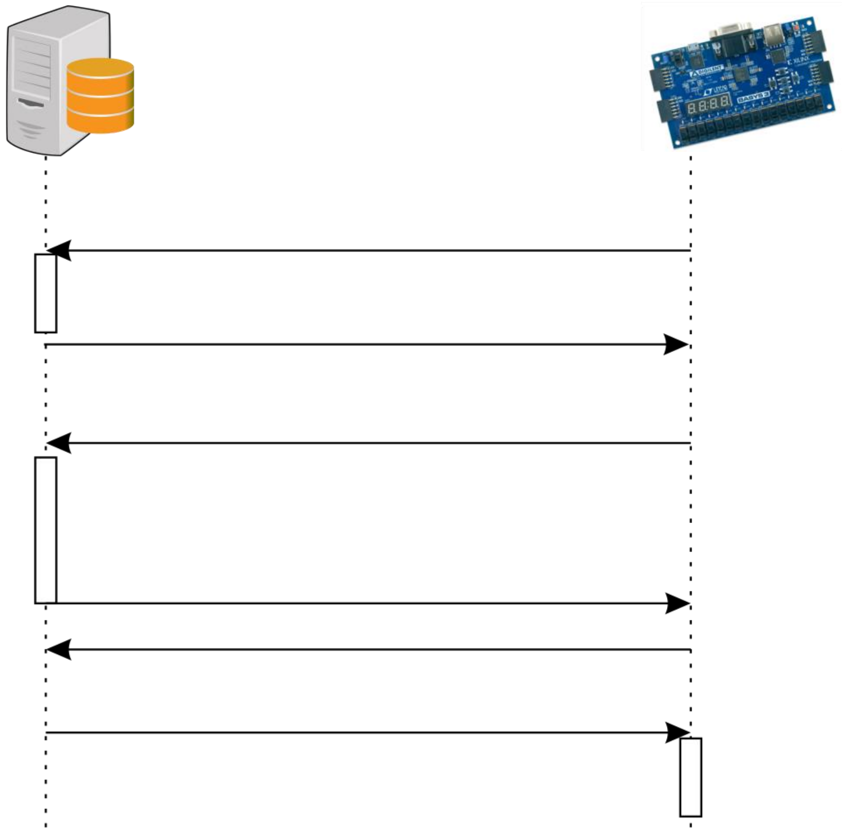
Our Approach

Update Protocols

- Metadata exchange
- Find good trade-offs
  - Generic updates x Client processing

Metadata

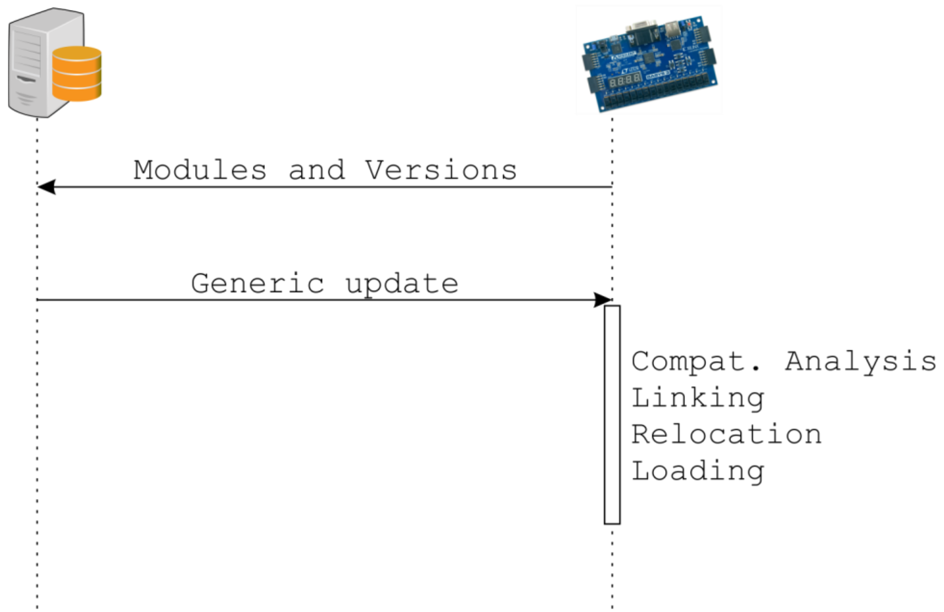
- List of installed modules and respective versions
- Global symbol table
- Memory layout
- Execution models



Our Approach

Metadata Location

- Server
- Client
  - List of installed modules and respective versions
  - Global symbol table
  - Memory layout
  - Execution models

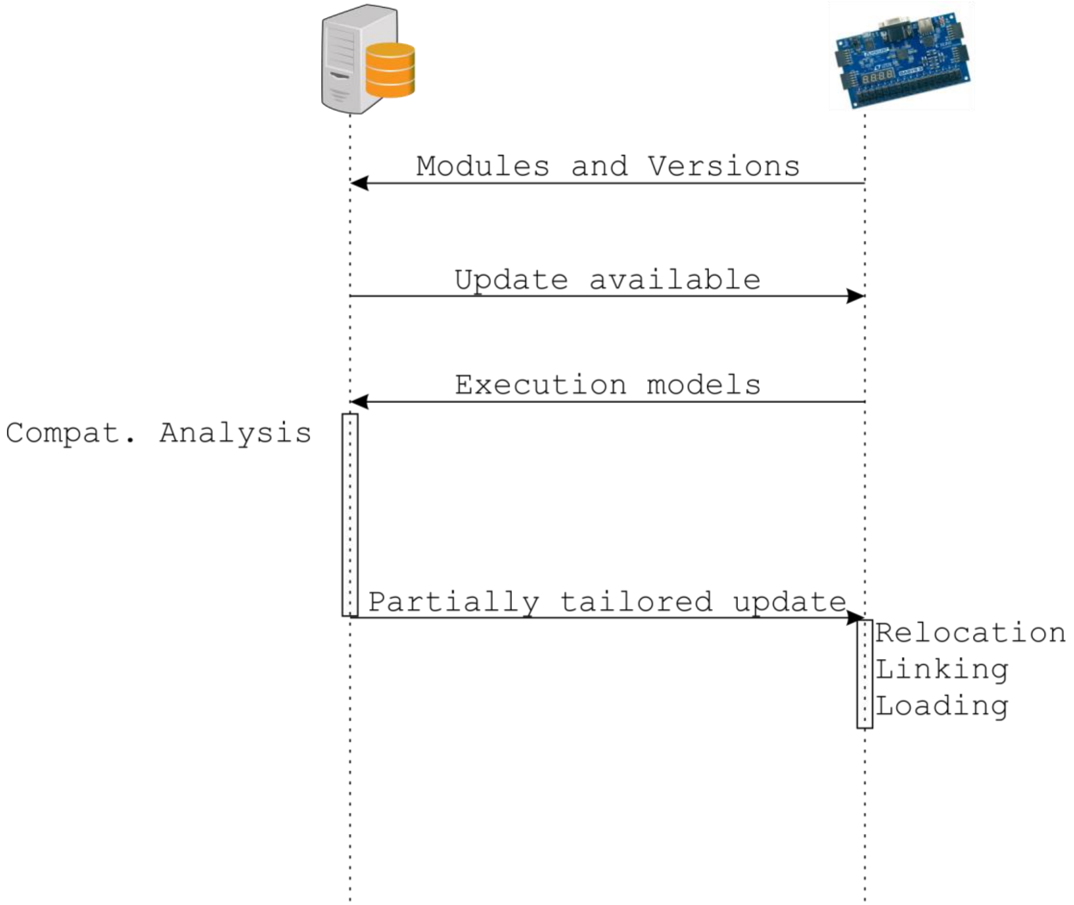


Our Approach

Metadata Location

- Server

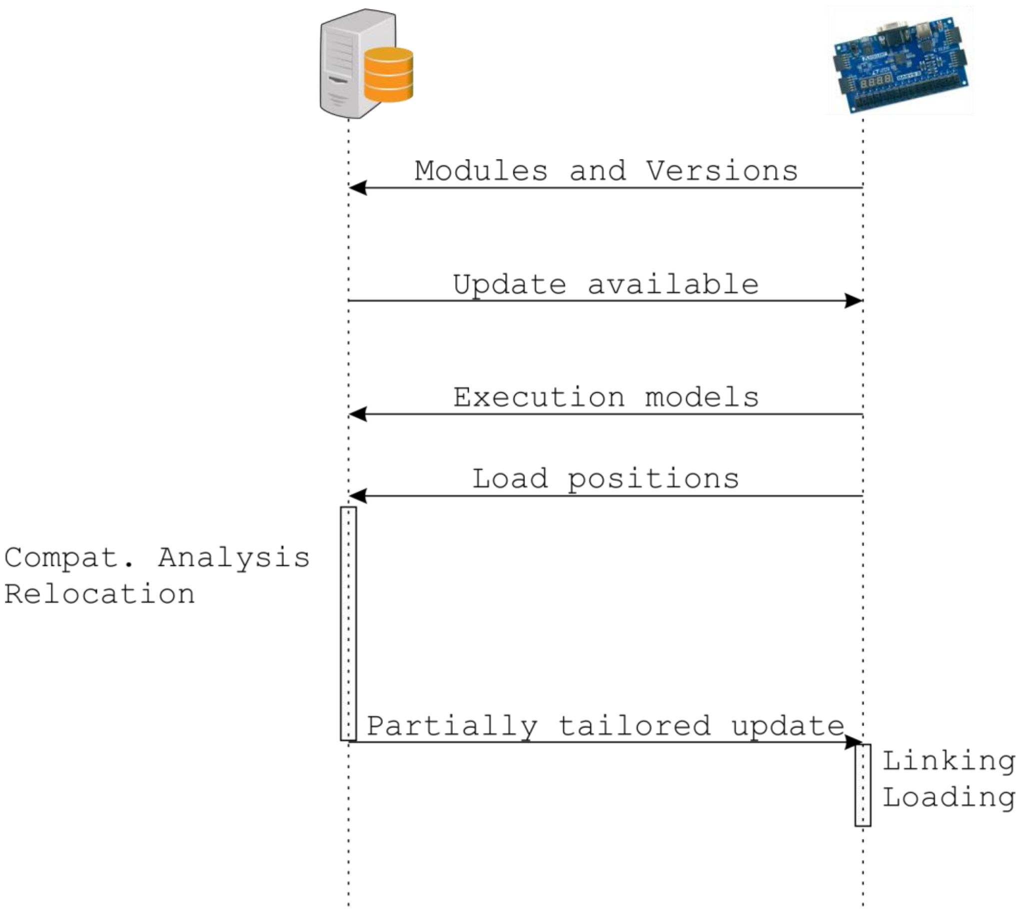
- Client
  - List of installed modules and respective versions
  - Global symbol table
  - Memory layout
  - Execution models



Our Approach

Metadata Location

- Server
- Client
  - List of installed modules and respective versions
  - Global symbol table
  - Memory layout
  - Execution models

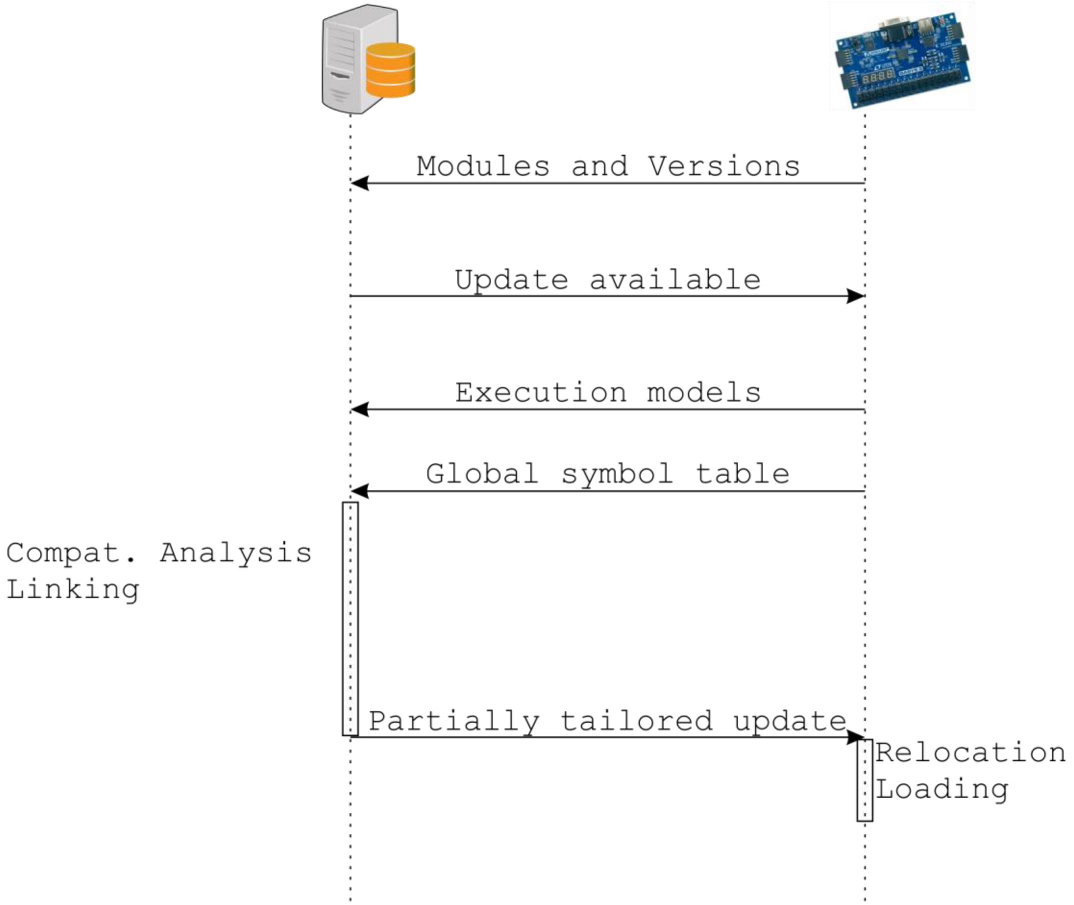


Our Approach

Metadata Location

- Server

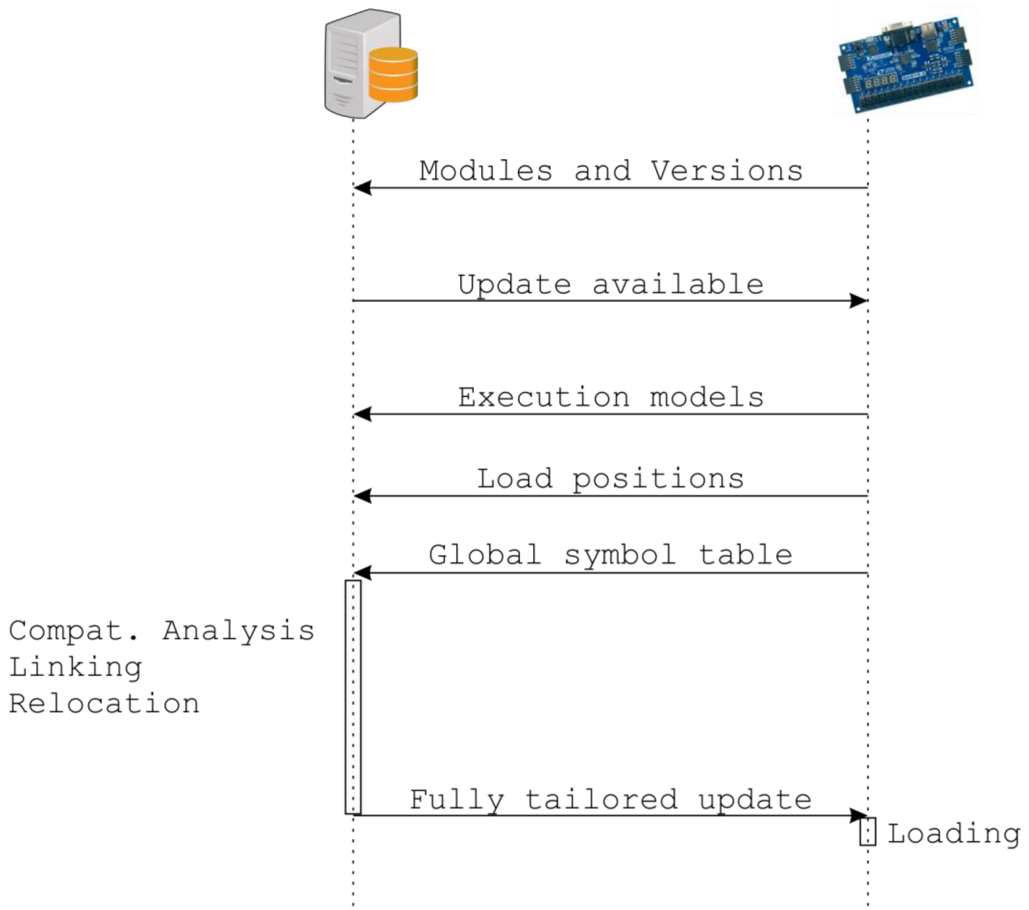
- Client
  - List of installed modules and respective versions
  - Global symbol table
  - Memory layout
  - Execution models



Our Approach

Metadata Location

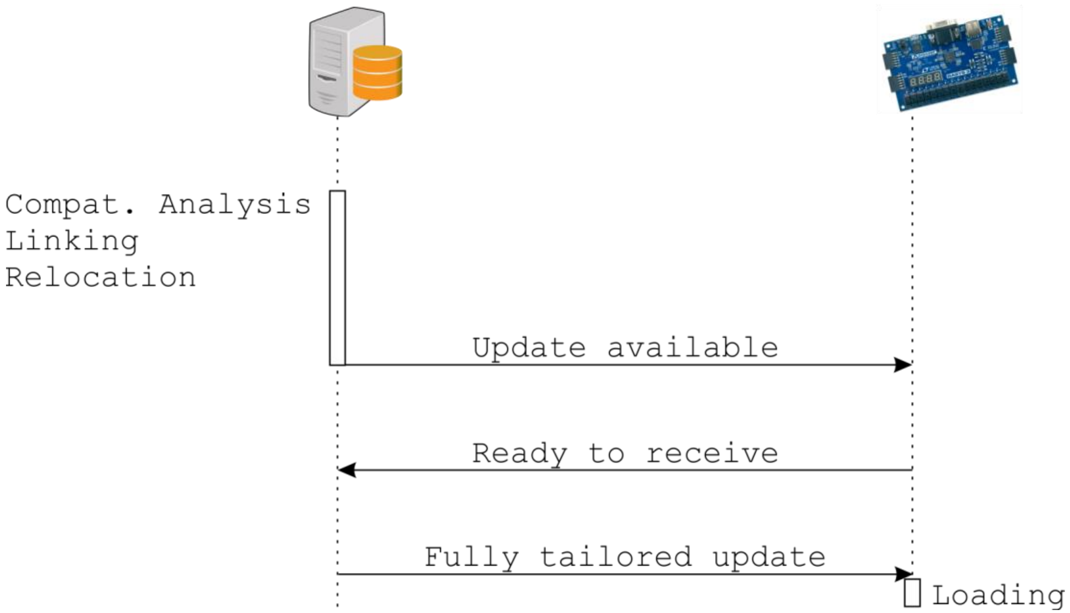
- Server
- Client
  - List of installed modules and respective versions
  - Global symbol table
  - Memory layout
  - Execution models



Our Approach

Metadata Location

- Server
  - List of installed modules and respective versions
  - Global symbol table
  - Memory layout
  - Execution models
- Client



Our Approach

- Investigate overhead with diverse update protocols.
- Define what performance classes of devices will support given protocols.

Thank you!

Leandro Batista Ribeiro
lbatistaribeiro@tugraz.at

16 November 2017

Institute of Technical Informatics
Embedded Automotive Systems Group
Graz University of Technology

References

[1] R. K. Panta and S. Bagchi. *Hermes: Fast and energy efficient incremental code updates for wireless sensor networks*. In INFOCOM 2009, IEEE, pages 639-647.

[2] H. Shin and H. Chao. *Supporting Application-Oriented Kernel Functionality for Resource Constrained Wireless Sensor Nodes*. In MSN, Hong Kong, China, December 2006.

[3] A. Dunkels, N. Finne, J. Eriksson and T. Voigt. *Run-time dynamic linking for reprogramming wireless sensor networks*. In Proceedings of the 4th international conference on Embedded networked sensor systems, 2006, pages 15-28. ACM, 2006.

[4] W. Dong, C. Chen, X. Liu, J. Bu, and Y. Liu. *Dynamic linking and loading in networked embedded systems*. In Mobile Adhoc and Sensor Systems, 2009. MASS'09. IEEE 6th International Conference on, pages 554-562.

[5] J. Koshy and R. Pandey. *Remote incremental linking for energy-efficient reprogramming of sensor networks*.

In Wireless Sensor Networks, 2005. Proceedings of the Second European Workshop on, pages 354-365.

References

[6] W. Dong, C. Chen, J. Bu, and W. Liu. *Optimizing relocatable code for efficient software update in networked embedded systems*. ACM Transactions on Sensor Networks (TOSN), 11(2):22, 2015.

[7] N. Tsiftes, A. Dunkels, and T. Voigt. *Efficient sensor network reprogramming through compression of executable modules*. In Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON'08. 5th Annual IEEE Communications Society Conference on, pages 359-367.

[8] J. Montgomery. *A model for updating real-time applications*. Real-Time Systems, 27(2):169-189, 2004

[9] H. Seifzadeh, A. A. P. Kazem, M. Kargahi, and A. Movaghar. *A method for dynamic software updating in real-time systems*. In Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on, pages 34-38