Approaches to Ultra Long-Term System Maintenance

Embedded Linux Conference Europe 2016

Prof. Dr. Wolfgang Mauerer
Siemens AG, Corporate Research and Technologies
Smart Embedded Systems
Corporate Competence Centre Embedded Linux
Overview

1  Introduction

2  Aspects of Long-Term Maintenance
   - Architectural Characteristics
   - Threats and Risks

3  Technical Aspects
   - Payload Software
   - Developing, Building and Testing
   - In-Field Strategy

4  Backporting & Processes
   - Backporting: Conceptual and Technical Issues
Outline

1. Introduction

2. Aspects of Long-Term Maintenance
   - Architectural Characteristics
   - Threats and Risks

3. Technical Aspects
   - Payload Software
   - Developing, Building and Testing
   - In-Field Strategy

4. Backporting & Processes
   - Backporting: Conceptual and Technical Issues
Disclaimer

- Many statements: Extremely obvious
- Realisation: Quite remote for many problematic appliances
- Quantification: Astonishingly hard...
Introduction I

Consumer Electronics
- Mobile Phones, Notebooks, Tablets, ... 
- Entertainment systems (Radio, TV, DVD/Blue Ray, ...) 
- Ovens, Washing Machines, Home Control/Automation

Industrial Systems
- Medical devices
  - Computed tomography, X-Ray Imaging, Ultrasound, ...
- Infrastructure
  - Gas, Power, Water supply 
  - Powerstations and transformers 
  - Traffic lights, park space management 
- Mobility
  - Planes, trains, automobiles, mars rovers, space stations
Fundamental questions

- Is long-term maintenance reasonable/doable?
- System architecture for LTM?
## Innovation Cycles I

<table>
<thead>
<tr>
<th>Lifespans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer devices: 2-5 years</td>
</tr>
<tr>
<td>Mobility: 5-20 years</td>
</tr>
<tr>
<td>Industrial: 10-30 years</td>
</tr>
<tr>
<td>Infrastructure: 30-80 years (and up!)</td>
</tr>
</tbody>
</table>

### All domains: Linux, of course!

- Long-life requirements not restricted to industrial appliances!
- IoT, smart home, connected devices: Longevity requirements pervade everyday devices
- Short lifespans: Exception, not rule!
Innovation Cycle III

Fundamental Questions
- Risks and benefits of updates?
- How to restrict updates to (isolated) areas?
- How to avoid updates?

Beyond components
- Questions not addressed by simply using LTS components/distros
- LTM: Architectural issue
- LTM: Mindset issue

Some field observations/bogus assumptions
- All components can be upgraded in-field
- Updates fix more problems than they create
- Upstream integration always reduces maintenance effort
- Long-term component versions solve maintenance problems
Innovation Cycle III

Fundamental Questions

- Risks and benefits of updates?
- How to restrict updates to (isolated) areas?
- How to avoid updates?

Beyond components

- Questions not addressed by simply using LTS components/distros
- LTM: Architectural issue
- LTM: Mindset issue

Some field observations/bogus assumptions

- All components can be upgraded in-field
- Updates fix more problems than they create
- Upstream integration always reduces maintenance effort
- Long-term component versions solve maintenance problems
Outline

1. Introduction

2. Aspects of Long-Term Maintenance
   - Architectural Characteristics
   - Threats and Risks

3. Technical Aspects
   - Payload Software
   - Developing, Building and Testing
   - In-Field Strategy

4. Backporting & Processes
   - Backporting: Conceptual and Technical Issues
Appliance Architecture

Long-term maintenance vs. periodic re-building

- Fixed (trusted) ↔ arbitrary payload software
- Isolated ↔ universally accessible
- Hardware stability ↔ variance
- Fixed hardware ↔ extensibility (e.g., USB)
- Verification and safety requirements
- Cost sensitivity (core payload inside virtual environments?)

Software aspects

- System base software
- Payload software + architecture
Appliance Architecture

Long-term maintenance vs. periodic re-building

- Fixed (trusted) ↔ arbitrary payload software
- Isolated ↔ universally accessible
- Hardware stability ↔ variance
- Fixed hardware ↔ extensibility (e.g., USB)
- Verification and safety requirements
- Cost sensitivity (core payload inside virtual environments?)

Software aspects

- System base software
- Payload software + architecture
Appliance Architecture

Long-term maintenance vs. periodic re-building

- Fixed (trusted) ↔ arbitrary payload software
- Isolated ↔ universally accessible
- Hardware stability ↔ variance
- Fixed hardware ↔ extensibility (e.g., USB)
- Verification and safety requirements
- Cost sensitivity (core payload inside virtual environments?)

Software aspects

- System base software: Little/no control
- Payload software + architecture: Full control ⇒ LTM Focus!
Threats and Risks

What should LTM prevent in your case?

- Device stops working
- Device faults cannot be repaired/debugged
- Device can be influenced from outside
- Device does not meet changed expectations (functionality, interoperability, ...)

Response catalogue

- Ignore issues (can be reasonable, on rare occasions)
- Replace device (HW + SW; component)
- Modify SW
Threats and Risks

What should LTM prevent in your case?

- Device stops working
- Device faults cannot be repaired/debugged
- Device can be influenced from outside
- Device does not meet changed expectations (functionality, interoperability, ...)

Response catalogue

- Ignore issues (*can* be reasonable, on rare occasions)
- Replace device (HW + SW; component)
- Modify SW ← *case of interest*
Outline

1. Introduction

2. Aspects of Long-Term Maintenance
   - Architectural Characteristics
   - Threats and Risks

3. Technical Aspects
   - Payload Software
   - Developing, Building and Testing
   - In-Field Strategy

4. Backporting & Processes
   - Backporting: Conceptual and Technical Issues
Software Engineering Considerations

- Deliver maintainable software/architecture in the first place
  - Minimise cross-cutting issues
  - Harmonise technical and social organisation
  - Meaningful and reproducible history
- Think *three times* before connecting systems to networks; *then think three more times*
- “Translator” with domain and (base component) community knowledge
- Make components (run-time) replaceable; prefer userland to kernel
Payload Software

Software Engineering Considerations

- Deliver maintainable software/architecture in the first place
  - Minimise cross-cutting issues
  - Harmonise technical and social organisation
  - Meaningful and reproducible history
- Think \textit{three times} before connecting systems to networks; \textit{then think three more times}
- “Translator” with domain and (base component) community knowledge
- Make components (run-time) replaceable; prefer userland to kernel
Reproducible Builds

- Produce binaries...20 years after initial launch
  - Payload application + modifiable system components
  - Preserve base component binaries
- Documentation of (seemingly trivial) details essential
- Documentation availability (hardcopy *is* a serious alternative)
- Avoid custom build systems
## Development & Building I

### Reproducible Builds

- Produce binaries...20 years after initial launch
  - Payload application + modifiable system components
  - Preserve base component binaries
- Documentation of (seemingly trivial) details essential
- Documentation availability (hardcopy *is* a serious alternative)
- Avoid custom build systems

### Source Code

- Availability of source code + history (e.g., Bitkeeper...)
- Component states + *local provision* of dependencies
- Includes build infrastructure!
Reproducible Builds

- Produce binaries...20 years after initial launch
  - Payload application + modifiable system components
  - Preserve base component binaries
- Documentation of (seemingly trivial) details essential
- Documentation availability (hardcopy is a serious alternative)
- Avoid custom build systems

Tool Chain

- Cross-Building: (subtle) dependencies!
- Isolate build environment in VM (strict freeze!)
- Bugs in ancient toolchain: Payload SW workarounds
- Eclipse etc.: harder...✓
Component Selection and Integration

- Consider cost of libraries
  - Dynamically changing dependencies (version requirement specs often unreliable)
  - Changes in components ⇒ (silent) breakage in library
- Distinguish between prototype and deliverable
  - Experiment with 17 machine learning algorithms
  - *Deploy* one (+ rewrite)
Development & Building II

Development prior to market release

- Develop against latest mainline state (rebasing preferred)
- *Avoid vendor BSPs*. Board support essential, not BSPs!
  - Only chance: *Prior* to purchasing $1.8 \times 10^{23}$ units
- System changes: Upstream first policy
- *Avoid* component modifications (socio-technical congruence)
  - Especially for features useless for upstream
- Minimise divergence between upstream state and product at release time
Five Recommendations

1. Avoid complex development environments and generated code
2. Avoid web technologies
3. Use convenience libraries judiciously
4. Avoid integration/consolidation; delegate communication/networking to separate entities
5. Document and automate excessively
Options for post-release development

### Rolling Development
- Continuous updates of (selected) base components
- Uncouple progress from distribution (e.g., after eol)
- Detect issues early, re-invent distribution wheel

### Distribution schedule
- `sudo apt-get upgrade`
- Requires support by distribution!
## Options for post-release development

### System schedule
- Update in appliance-specific intervals (periodic or irregular)
- Combine disadvantages of distribution and continuous updates

### Invariant base system
- Don’t update base system
- Payload application development only
- Requires (extremely) small attack surface/virtualised base system
Outline

1 Introduction

2 Aspects of Long-Term Maintenance
   - Architectural Characteristics
   - Threats and Risks

3 Technical Aspects
   - Payload Software
   - Developing, Building and Testing
   - In-Field Strategy

4 Backporting & Processes
   - Backporting: Conceptual and Technical Issues
Backporting I

Leverage LTSI kernel

- LTSI support period: Comprehensive coverage
- Post-LTSI: Backport only
  - Critical issues re/ attack surfaces
  - Orthogonal drivers/components
  - Feature not required during first 5 years ⇒ unlikely required in next decade(s)
  - Major change required (debugging, tracing etc.): Time for new release...

Backport patch stack

- Organise backports in proper orthogonal patch stack
- Rebase! Living organism, not code dump
Backporting II: What and when to backport

1.) What to backport

- Most upstream changes *do not* require back-porting
- *Selection* crucial
- Selection criteria *differ* depending on use case

Approaches

- Keyword filtering (possible for well-tended projects)
- Content/file/path based filtering (tremendous volume reduction)
- Manual review + long-term expert involvement necessary
### Backporting II: What and when to backport

**2.) When to backport**

- Proactively
- After incidents/bugs

**Simple criterion**

- # incidents > # backport regressions
- Historical data: No conclusive evidence
- Expert assessment required
The human touch

- Determine when *no* action is required
- Notify users/customers
Backporting IV

Goal: Simplify patch selection for everyone

- Fully automatic approach: unrealistic
- Avoid duplicated manual efforts

Wishlist: Improvements

- Maintenance classes (for patches), consistent across projects
  - Current schemes dating back to 70ies ⇒ survey!
- Applicability range (releases) annotation
  - Wide-spread use of automated approaches (backwards integration, testing)
- Extend use of semantic vs. text-based modifications
LTM best practices: Similar to proper (OSS-style) software development

System and application architecture: crucial

LTM: *not* rocket science, but still more art than science – quantitative data required!
Thanks for your interest!