Updating Nodes in Distributed Networks

Dejaeghere, Jules

Publication date:
2024

Document Version
Publisher's PDF, also known as Version of record

Updating Nodes in Distributed Networks

Jules DEJAEGHERE

University of Namur

Research setting

Updating nodes participating in a distributed network

- Many different node operators
- No central administrator
- Many different versions may be running at the same time

Deploying new features in the network is technically challenging

Current approach

Updating the node is left at the discretion of its operator. Not all operators apply the updates at the same speed.

- Repository policies slow down update distribution
- No universal auto-update system (OS dependent)
- Operators must follow communication channels to be informed of new versions

Drawbacks of the current approach

The current approach has some limitations, making it harder for the developers to deploy new features

- Developers must leave room for new features, introducing flexibility
- Flexibility can be used as an attack vector
- Developers do not have full control on the nodes running the software
- Different operating systems have different requirements for updates

Ideas for a new approach

Provide a **stable binary** using the **usual software distribution channels** but allow updates to be applied **during the execution** of the software. The live updates are distributed using a channel controlled by the developers directly.

UNIVERSITÉ DE NAMUR

CyberExcellence project, grant 2110186 funded by the SPW Recherche

Requirements for an improved solution

Main binary should be stable

- Major updates to the main binary should follow OSes release cycle
- Default implementation is provided for all the features (may be a no-operation)

Updates are part of normal operation

- Updating should not require external scripts
- Update process should be platform independent
- Updates should happen automatically

Updates are hot swappable

- New code is loaded at runtime
- No need for admin to login

Updates may fail

- The core software can unload failing updates
- Rollback to previous version is automatic

Revisiting software architecture and execution

Architecture of a typical program

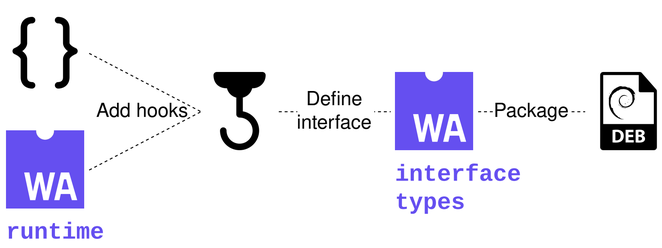
1. The *core software* comes with a default implementation of all the features
   - Contains hooks (places where updates can be applied)
   - Is the most stable part of the application
   - Embeds a WebAssembly runtime
2. Updates are WebAssembly modules
   - Updates are attached to a hook in the core

Execution of a typical program

1. When the *core software* reaches a hook
   - Check if an update module is available (locally)
   - Yes: execute the module
   - Else: execute the default implementation code
2. When a new update is published by the developers
   - The core fetches the update module
   - When the hook is reached the next time, the new module is used

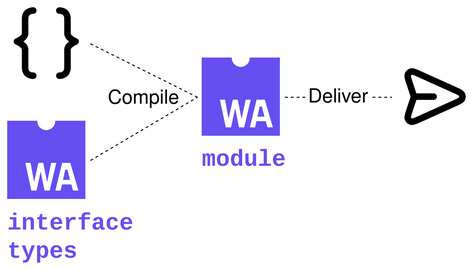
Workflow with the proposed framework

Main binary



1. Create the application the usual way
2. Define hooks (where future updates will be applied)
3. Define interface for the update modules
4. Define a distribution strategy for the updates

Update modules



1. Write the update module (compliant with the interface)
2. Compile the module to WebAssembly
3. Deliver the update module

Ongoing and future work

Implement the framework as a Rust crate

- Providing macros to define hooks
- Embedding a WebAssembly runtime to JIT compile the updates
- Providing primitives for update distribution

Extend the framework

- Other software may benefit from the framework
- Add nice-to-have features: auto-rollback, third-party updates

Example of use-case: updating Tor relays

Context

- Around 7500 relays
- No central administration

Benefits of the update modules

- Fast to distribute (see Figure 1)
- No protocol tolerance needed
- Automatic update
- JIT compilation: no downtime

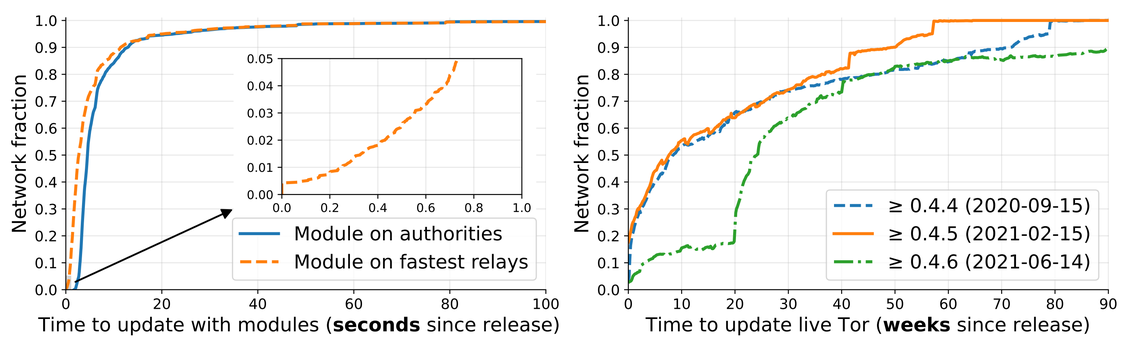


Figure 1. Update module propagation within a simulated network (left). Tor's situation based on historical metrics (right). Note the units.