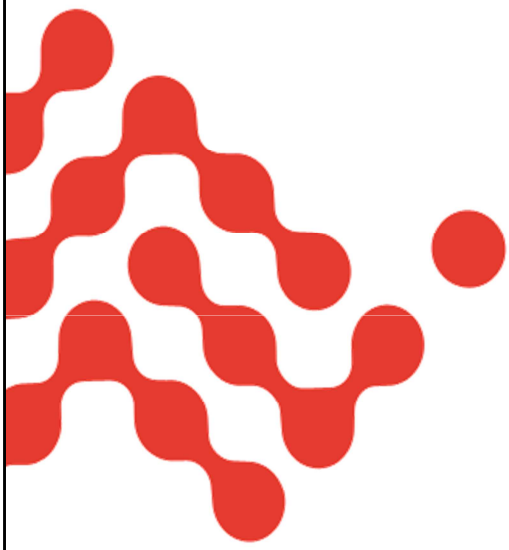
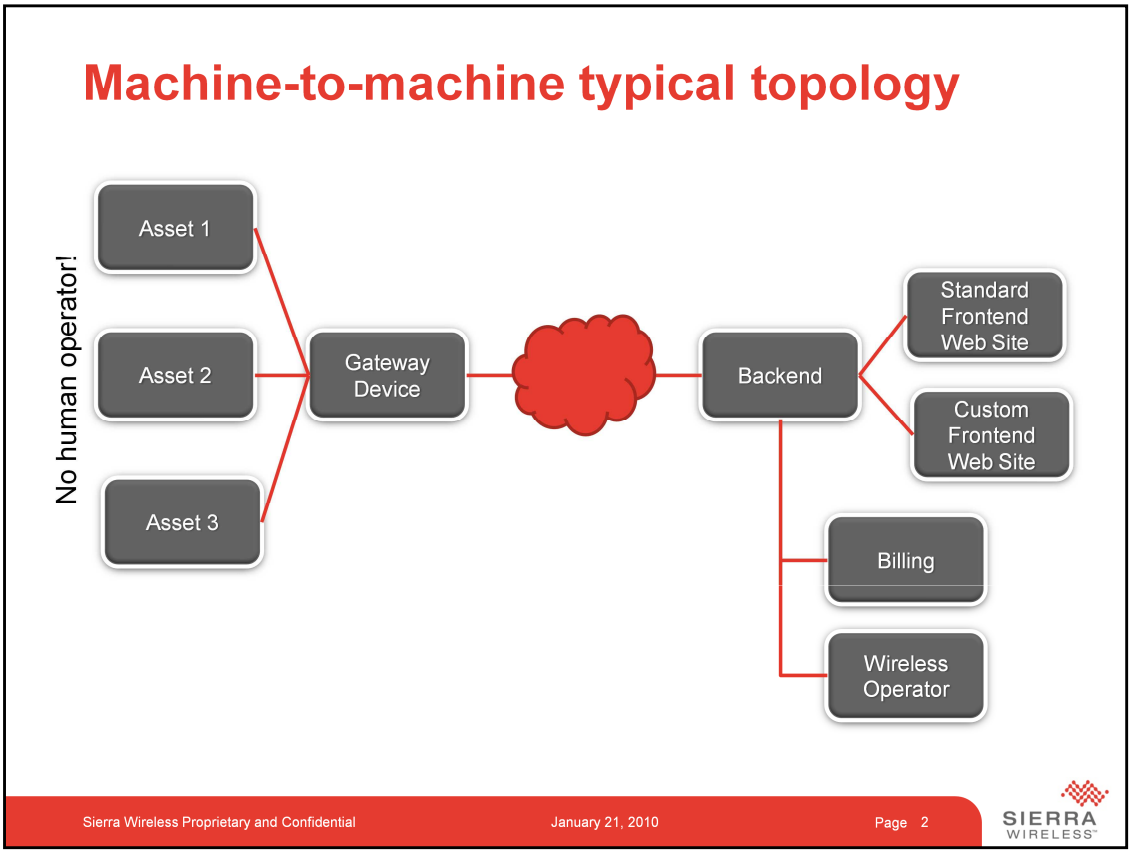
M2M embedded development with Lua

Lua Workshop '11



SIERRA WIRELESS™



Description of a typical M2M chain:

•An asset is a machine which needs to communicate over the net, without human operator intervention.

•There can be one or many assets; one of them used as a network gateway.

•Communication generally happens over the air, to preserve mobility and/or simplify logistics.

•A backend keeps asset info always available, presents them to 3rd party services and front-end.

•Frontend can be custom, but doesn't have to: we provide a sane and functional default, which only requires a bit of config.

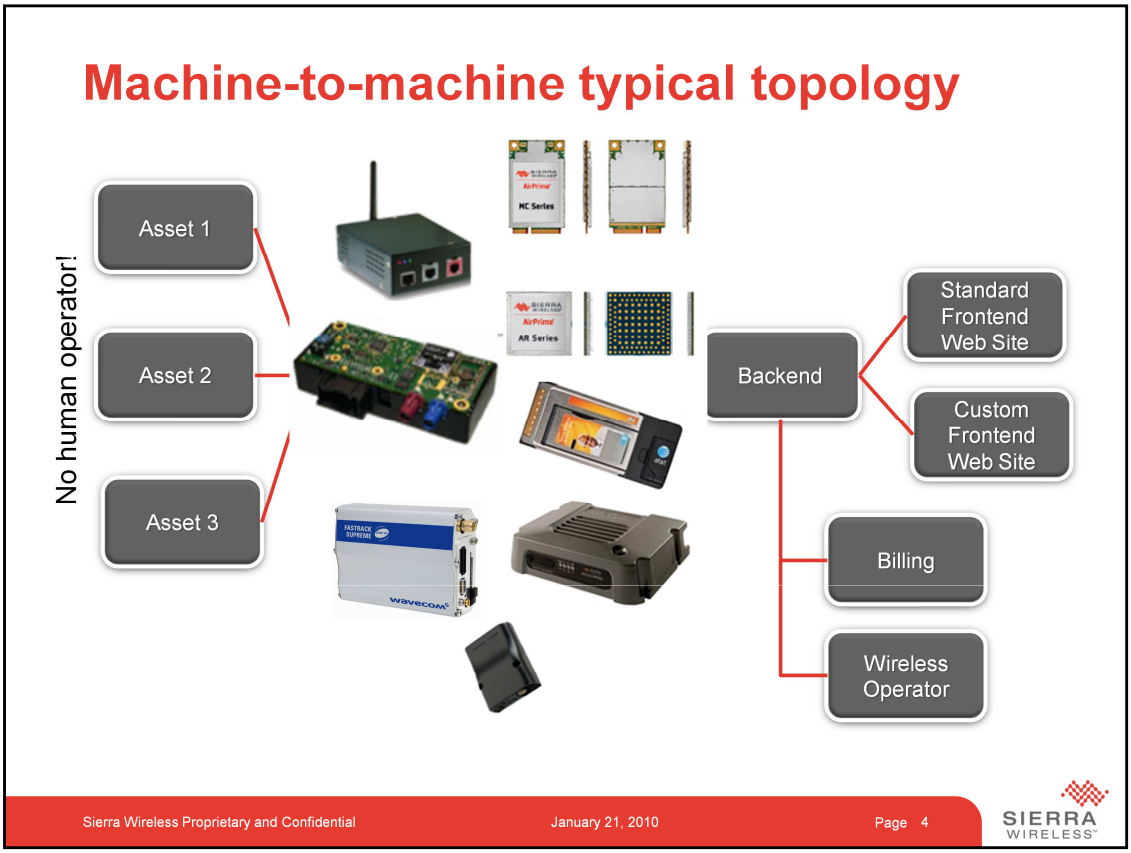
This chain requires many very different skills: hardware, antennas, embedded software, wireless network, server, UI, IT; extremely few companies will have all these skills at once, and assembling them from many subcontractors is no piece of cake either.

Some M2M asset examples

- Factory robots
- Wind turbines
- Public lights
- Automated billboards
- ATM
- Vending machines
- Utility meters
- EV charging stations
- Automated bicycle rental stations
- Vehicule or container tracking
- Alarm systems

SIERRA WIRELESS

Typical asset: valuable, in a remote position, needs to report events about itself (including failures) and/or it environmnent to the company owning it.



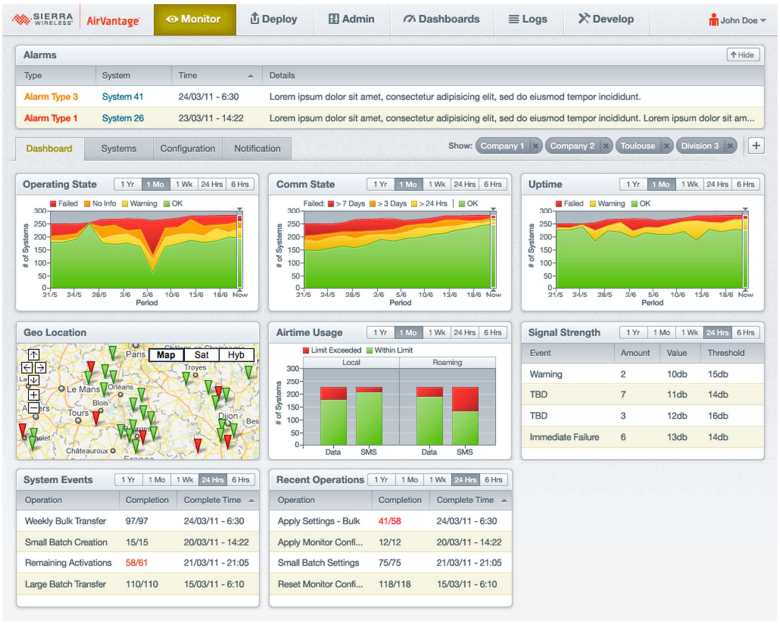
We're primarily a hardware company, we've plenty of HW options to offer: chips, PCB, reference designs, PCCards, routers, programmable modems… from a few dozen $ to many $100's, depending on integration, volume, I/O, programmability, CPU…

M2M standard front-end

SIERRA WIRELESS

We also offer a standard front-end, which can be heavily personalized. Based on widgets, role-based views. Fits most B2B needs: alerts reporting, large fleets management, data consolidation, provisionning, billing…

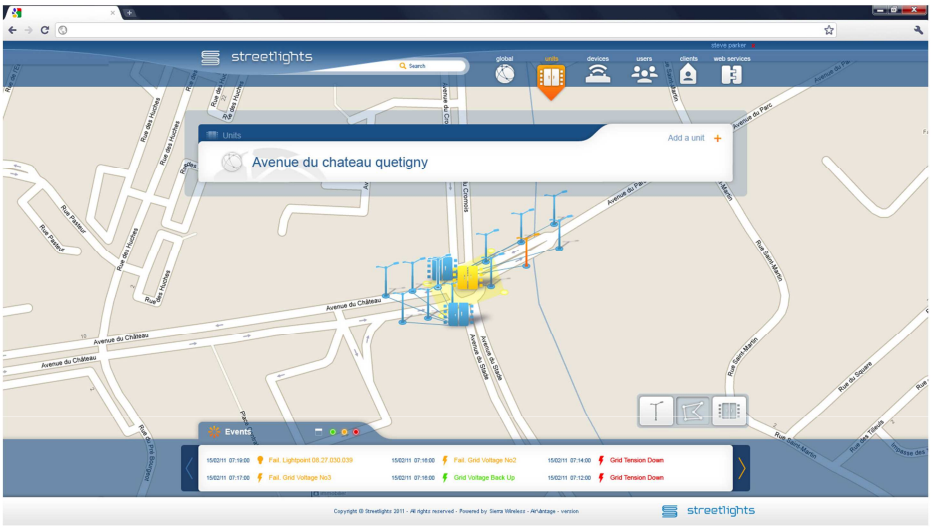
M2M standard front-end



SIERRA WIRELESS

Anotherr view composed of standard widgets.

Street Lighting Custom Front-end

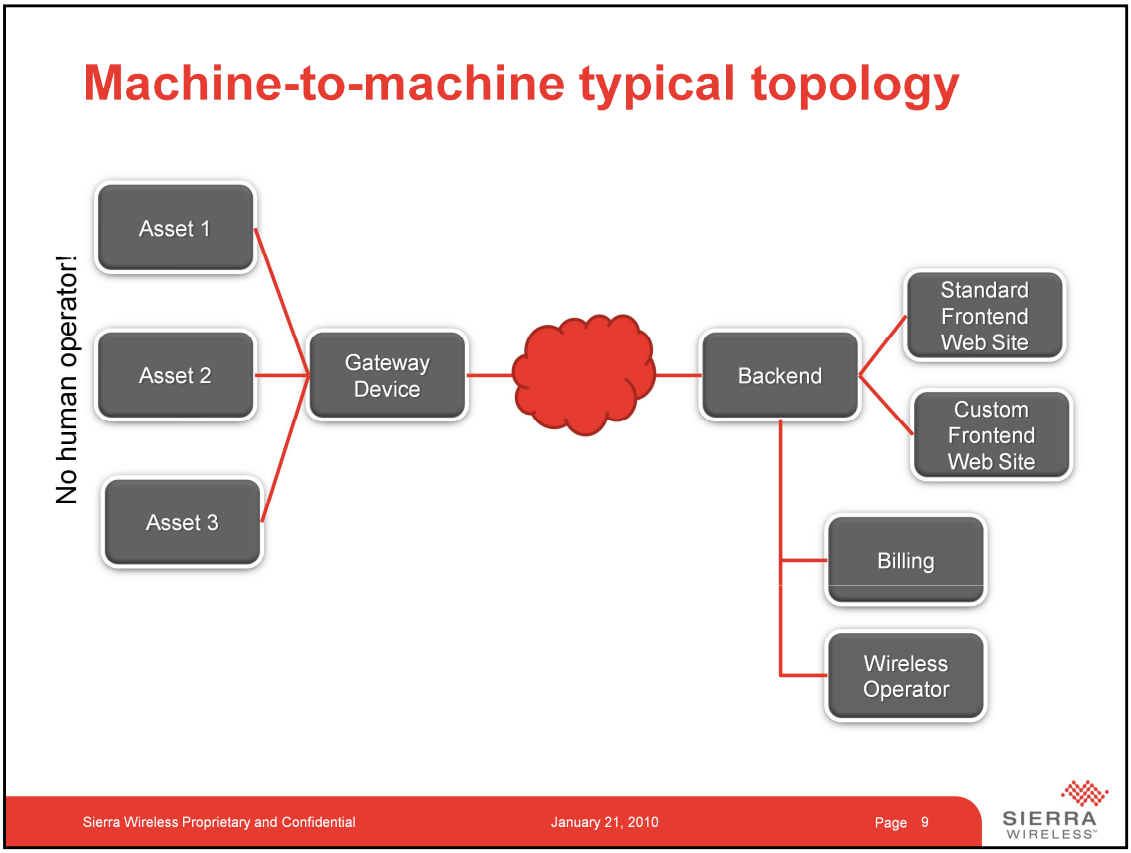


SIERRA WIRELESS

Example of a custom front-end, targetting non-professional operators (public streetlights maintenance).

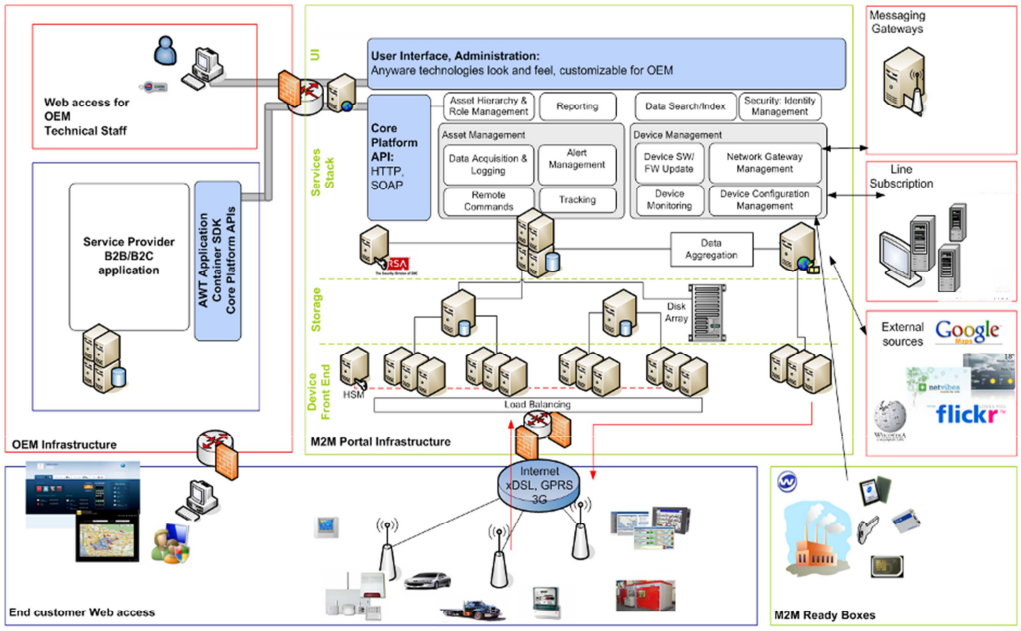
Home Automation Custom Front-end

A home automation system front-end, handling home alarm, heating systems, etc.



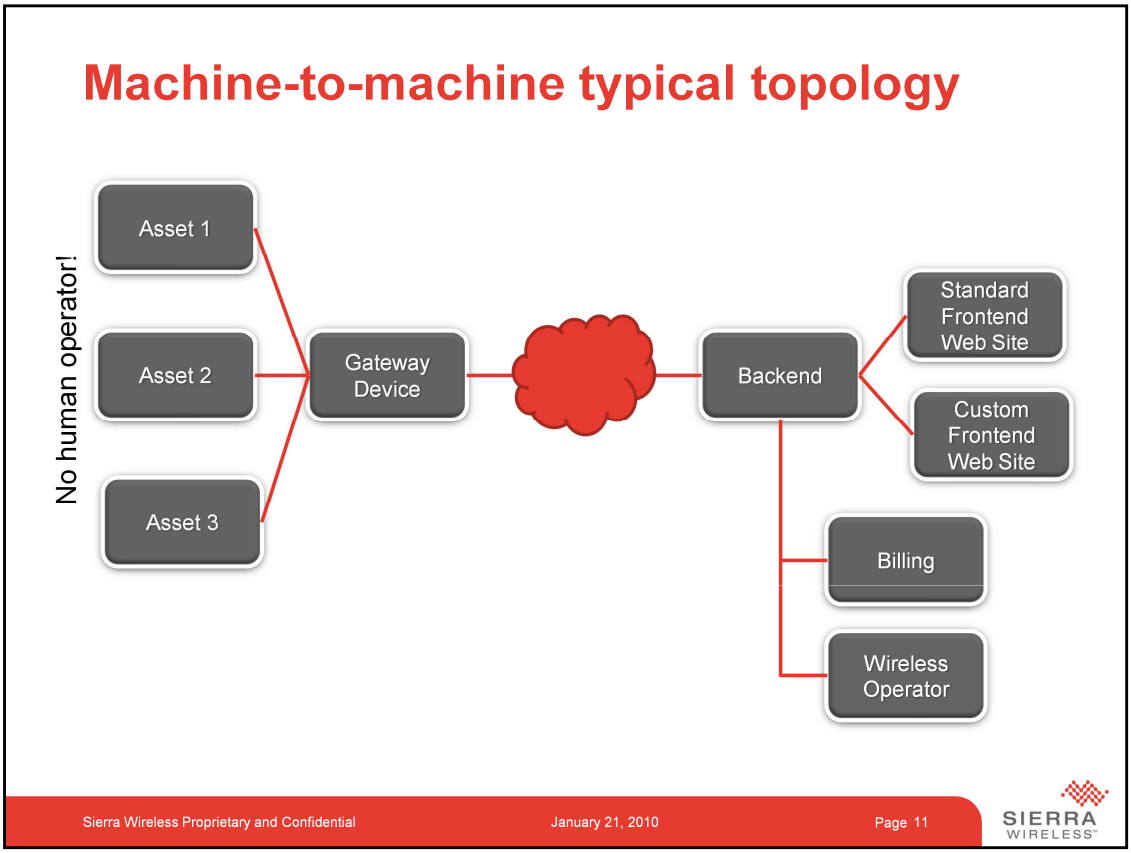
Back to the global view. The backend here is quite simplified…

A more realistic global view…



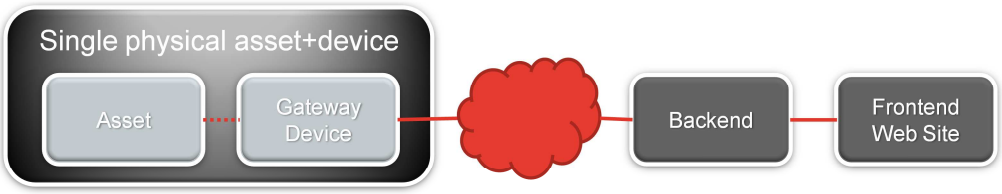
…but that's our business, not the customer's:

He's got a physical asset, a frontend (UI and/or REST API), and the latter faithfully represents the former. The rest ought to remain black magic.



Conversely, the situation is often simpler on the embedded side:
only one custom logical asset,
which is physically hosted on the gateway device's CPU.

Machine-to-machine typical topology



SIERRA WIRELESS

Only one custom logical asset, which is physically hosted on the device CPU.

Choosing Lua on the embedded side

Why not C?

- C is too low level, too long to debug, too hard to maintain
- Embedded C is not portable
- M2M changes faster than a C codebase could
- We're not in a hurry (we usually wait for GPRS!)
- We're not working on tiny µ-controllers anymore

What then?

- Python, Ruby, Javascript would have been fine for developers
- They're hard to port, resource-hungry
- Tedious to interface with C
- Only Lua and Scheme are technically realistic; Lua is way less scary
- It can still scare some middle-managers…

Key services

- Asset management:
  - Get data/events from assets
  - Push data/commands to assets
  - Persist/consolidate/report data
  - Follow communication policies
  - Monitoring services
- Device management: the device in an asset like any other
- Software update over-the-air: for device and external assets
- Applications management: modular scripts, Lua-only on simple devices
- Mediation: allowing server push with STUN, SMS…
- Communications: GPRS, SMS, Ethernet, router/DHCP, dialup, encryption, authentication…

SIERRA WIRELESS

What's useful from the non-developer customer's PoV.

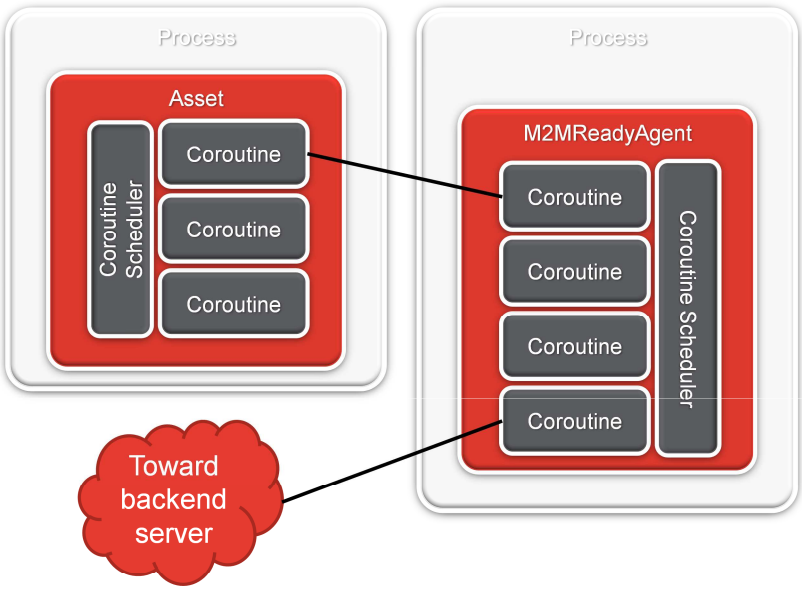
Key components

- Lua 5.1 (+ tweaks)
- Custom coroutine scheduler / synchro / IPC
- Serialization & streaming (Hessian)
- Luasocket, including LTN12 streaming
- More network protocols (Hessian, security, telnet, ntp…)
- Embedded protocols (modbus, AT commands, GPRS, OMA-DM…)
- Local DB / persistence / consolidation
- Advanced logging

SIERRA WIRELESS

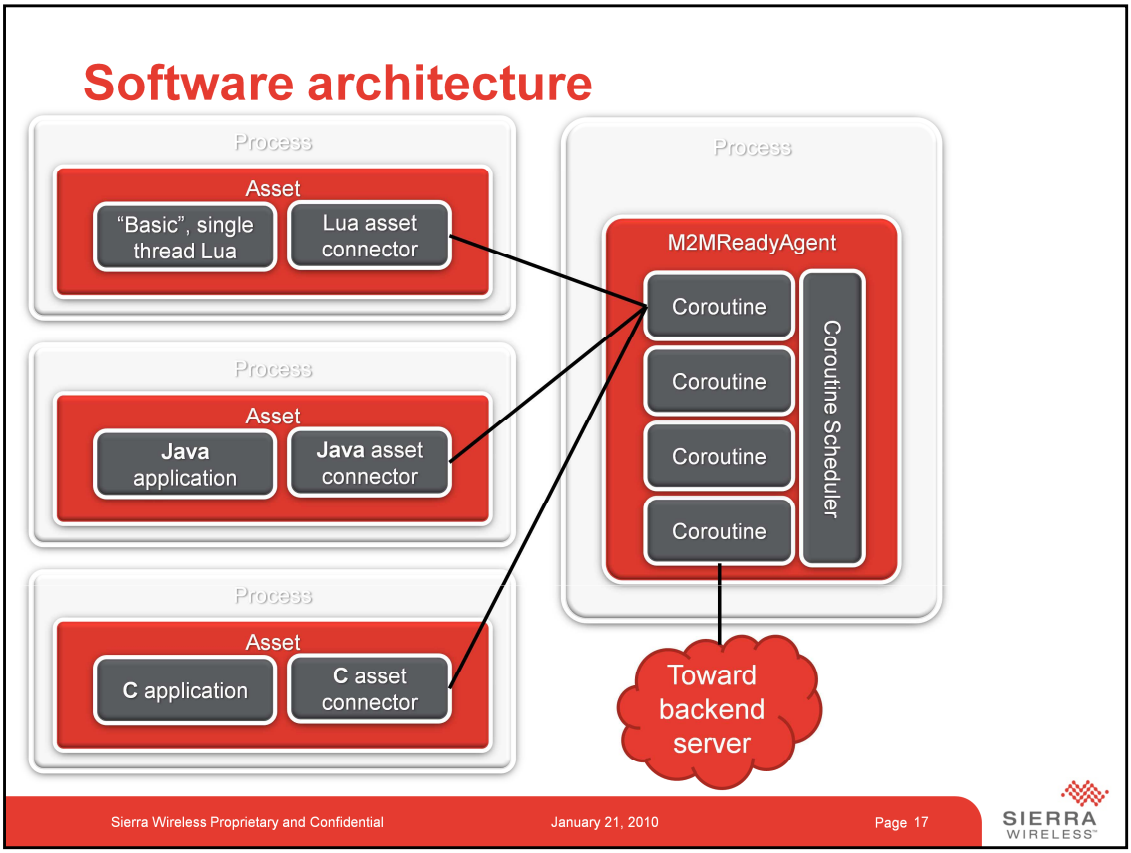
What the developers see.

Software architecture

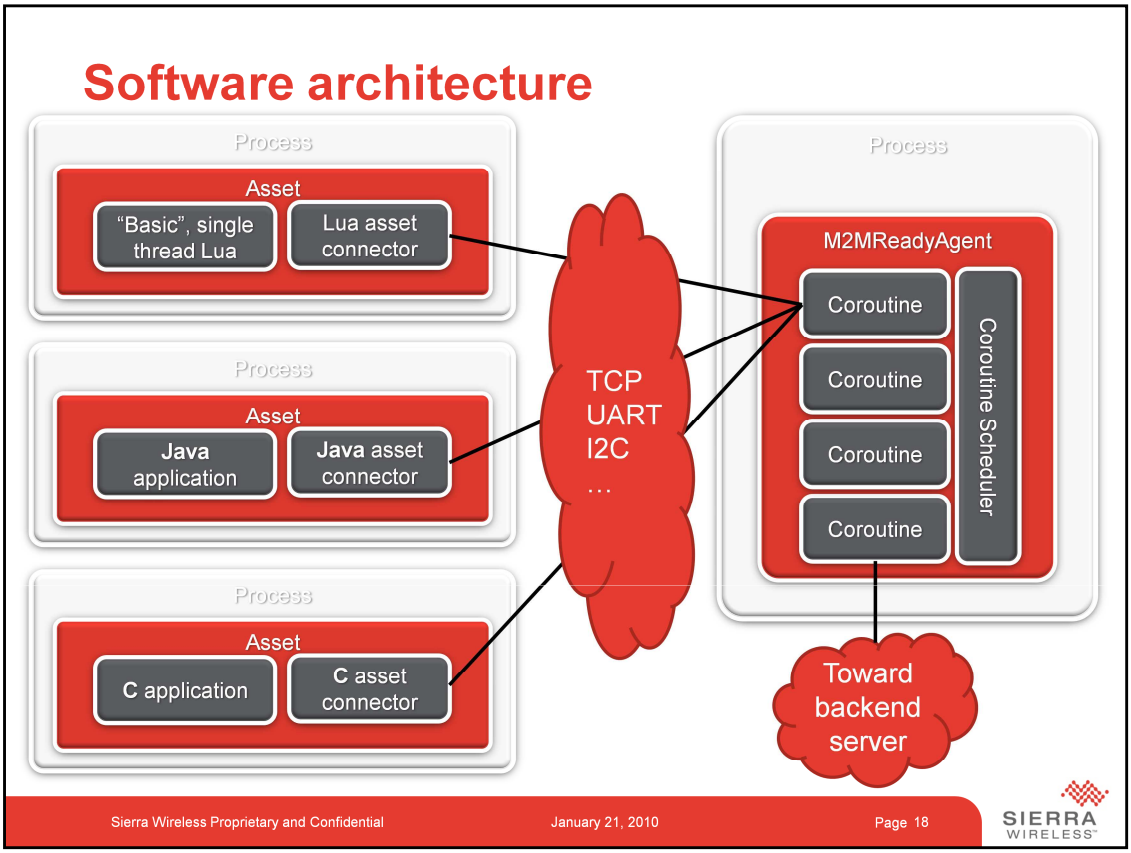


Basic architecture: two processes if possible, although we can run in a single one.

In any case, assets and agent only talk through a serialized RPC link. Keeps things cleaner/clearer.



One process per asset if possible, it avoids parasitic interactions, discourages avoidable couplings, and makes it easier to pinpoint bugs.



Assets don't have to run on the same CPU, the serialized RPC can go over physical links.

Coroutine-based multithreading

Multithreading built on a Lua coroutine scheduler:

- Not realtime (doesn't matter 99% of time)
- Not bug-prone (relative to other concurent systems)
- Flexible (easy task creation, synchronization)
- Portable on exotic / limited hardware
- Still occasionally requires mutexes

Multi-process when available: assets run separately

- Language-agnosticism for user code
- Limits the blame game, parasitic interactions

Concurrent execution is very surprisingly painless

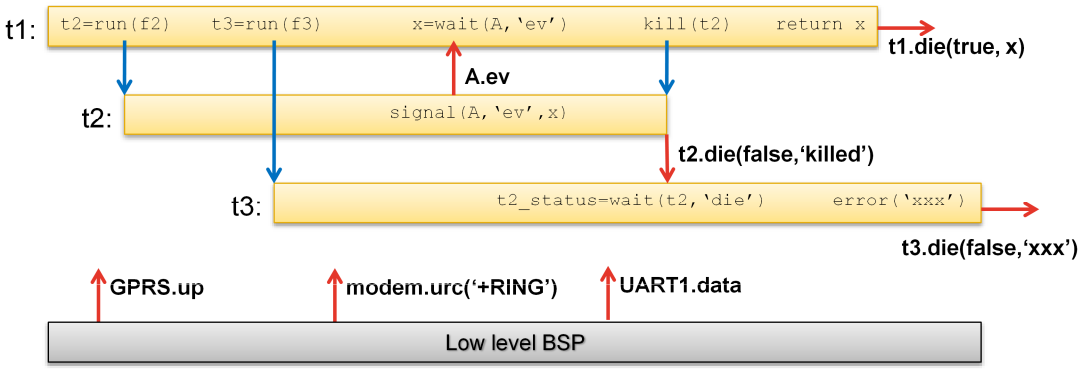
Scheduling

- Create or kill a task: `t=sched.run(function()…end)`, `sched.kill(t)`
- Send a signal: `sched.signal(emitter, event, [args…])`
  - Used for low level I/O
  - For inter-tasks synchronization
  - For extensibility
- Wait for a signal: `sched.wait(emitter, [timeout], events…)`
- React to signals: `sched.sighook()`, `sched.sigrun()`, `sched.sigrunonce()`

Written in pure Lua, with current state accessible as proc.tasks (never needed to convert to C).

The thread control and communication API.

Scheduling



Some simple examples of thread API events and calls.

Porting the scheduler

POSIX-based scheduler:

- Compute the due date of next timer event
- Get external inputs from file descriptors
- `select()` with timeout

Why not Copas?

- Really socket-centric
- Insufficient timer integration
- Lack of inter-thread communications
- Hard to port on exotic architectures

SIERRA WIRELESS

On POSIX, need to justify why we didn't reuse Copas, and why it wasn't an instance of the not-invented-here syndrome.

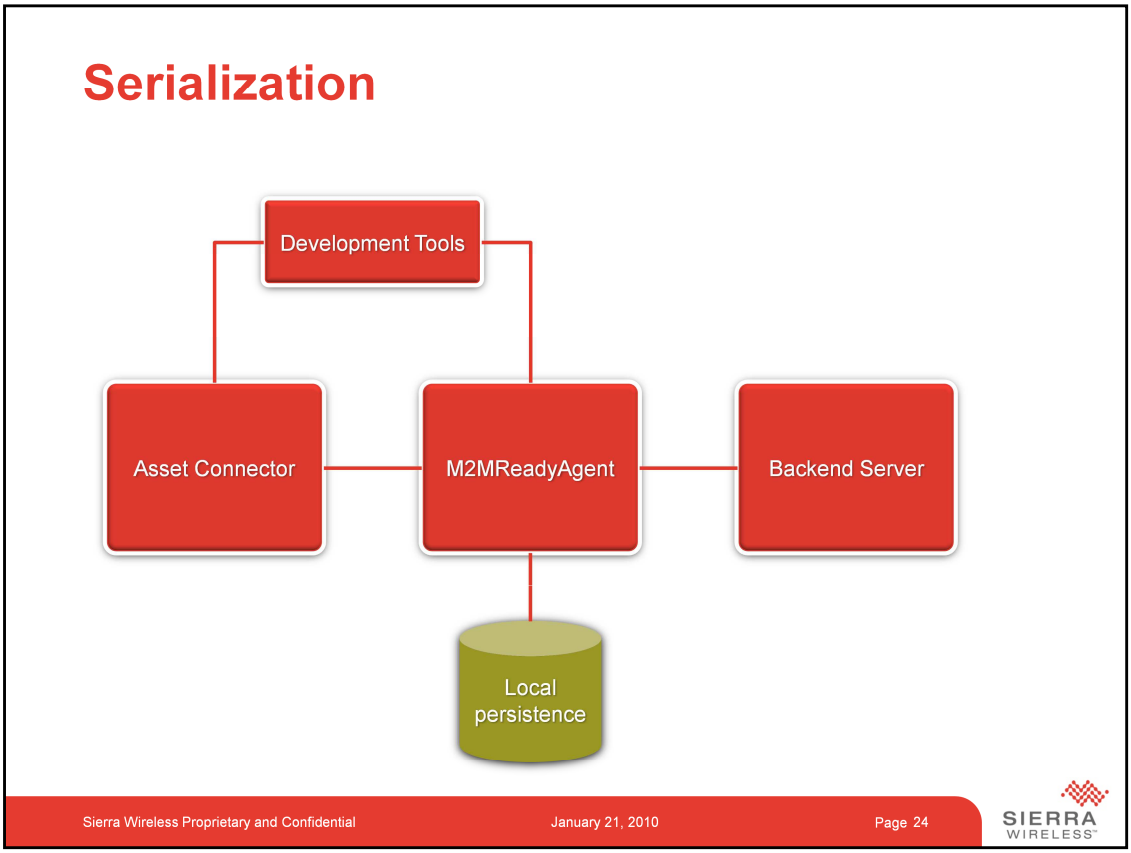
Porting the scheduler

Light OS scheduler:

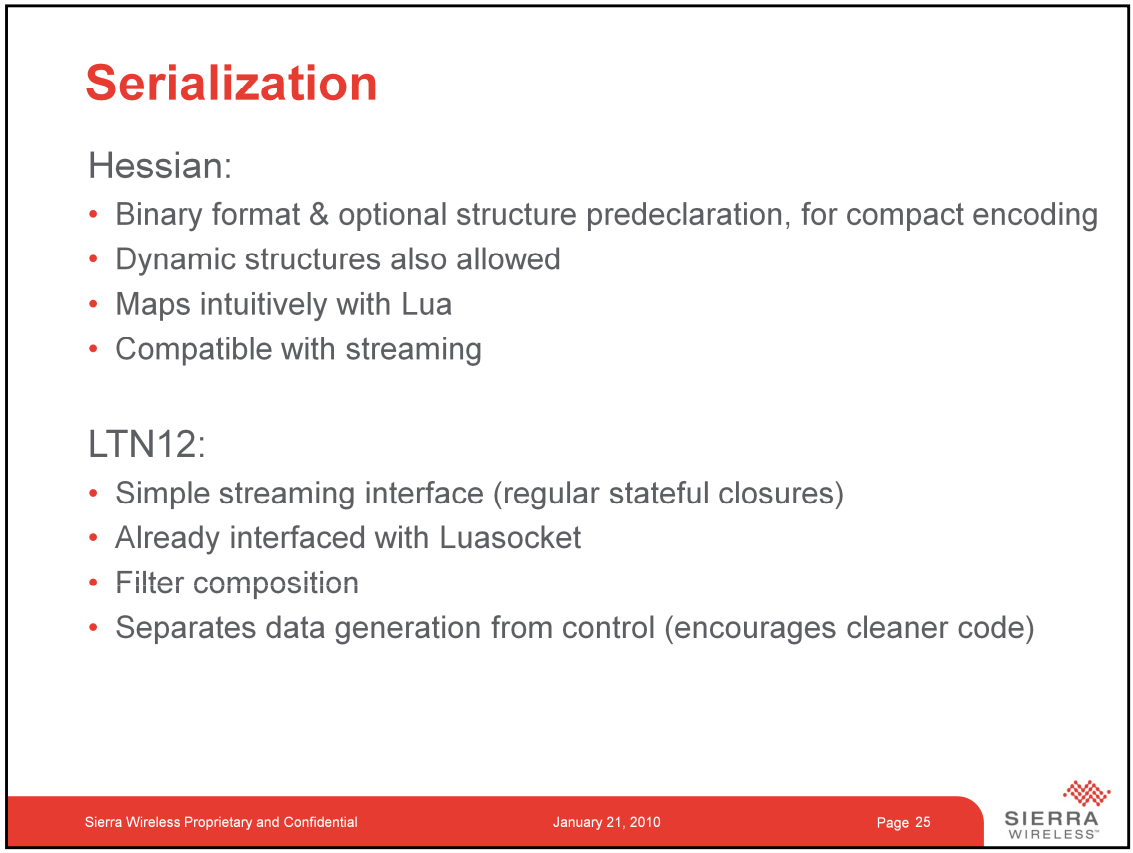
- Keep timer computations
- Based on a `luaW_signal()` C function to send low-level events to Lua and force scheduler wake-up
- Needs a `delayed_run(delay, lua_Cfunction)` primitive for timers.

Result:

- completely asynchronous, callback-driven at C level, with little OS requirements.
- Readable, multithreaded at Lua level.
- Additional constraint in C: no blocking function.



Many serialized communication channels; this is the dual of easy inter-thread communications: it enforces proper structuring at the scale where it matters. It also open interesting capabilities.

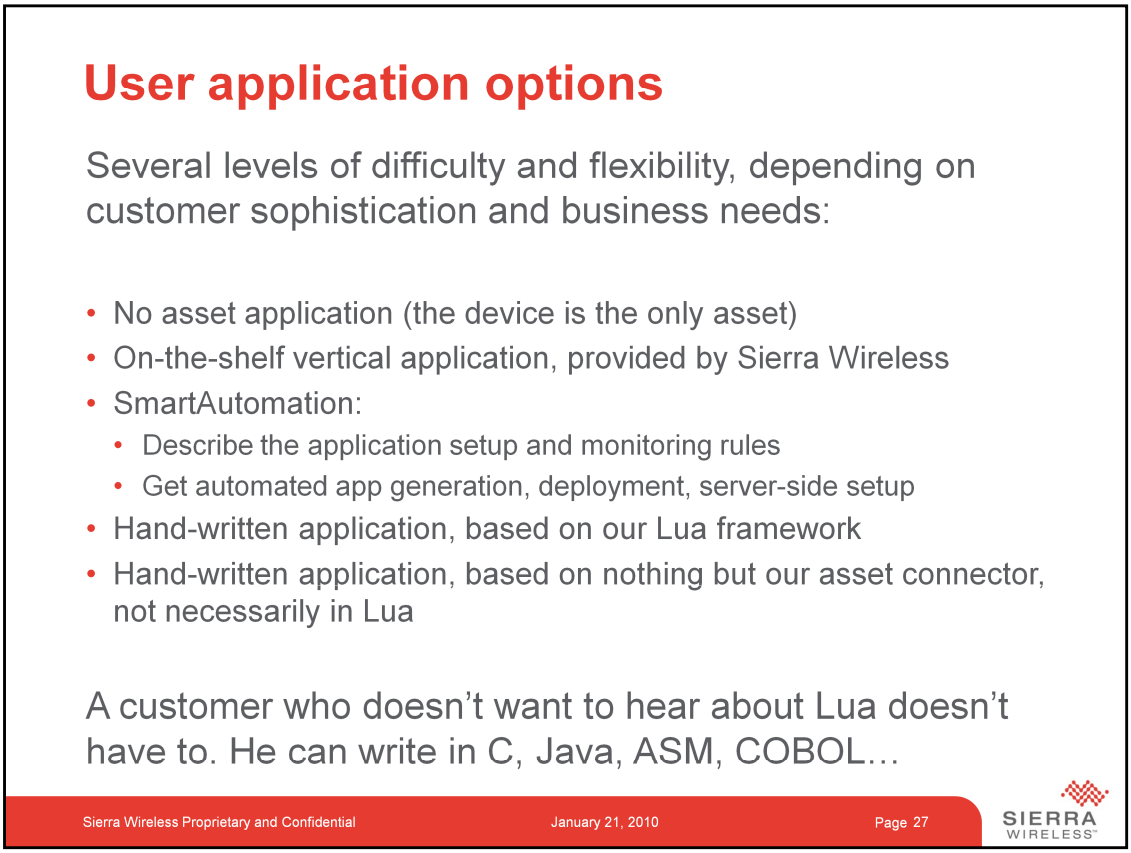


LTN12 really deserves more love than it gets. There are so many Lua libraries which would be made more usable by being exposed as LTN12 filters, developers ought to realize this.

LTN12

A basic chain:

Something more complex:

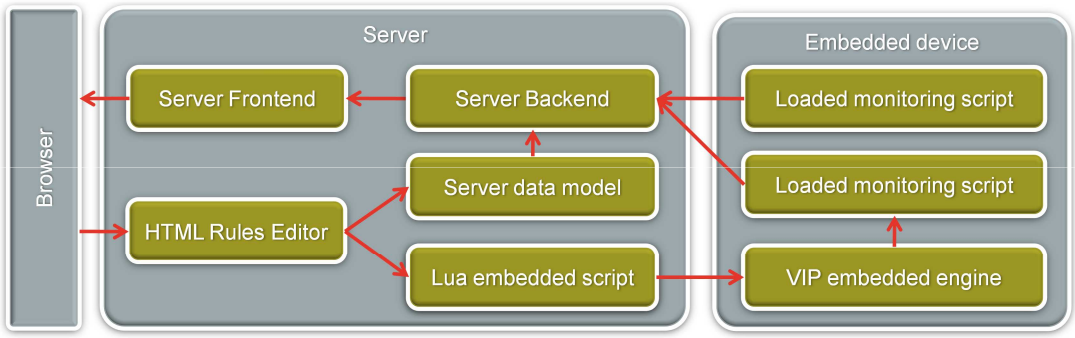


There's a continuum of options, from no development at all to completely developed from scratch on independent hardware. An interesting sweet spot is Smart Automation, only made possible because Lua makes code so flexible to write, use and move around.

SmartAutomation architecture

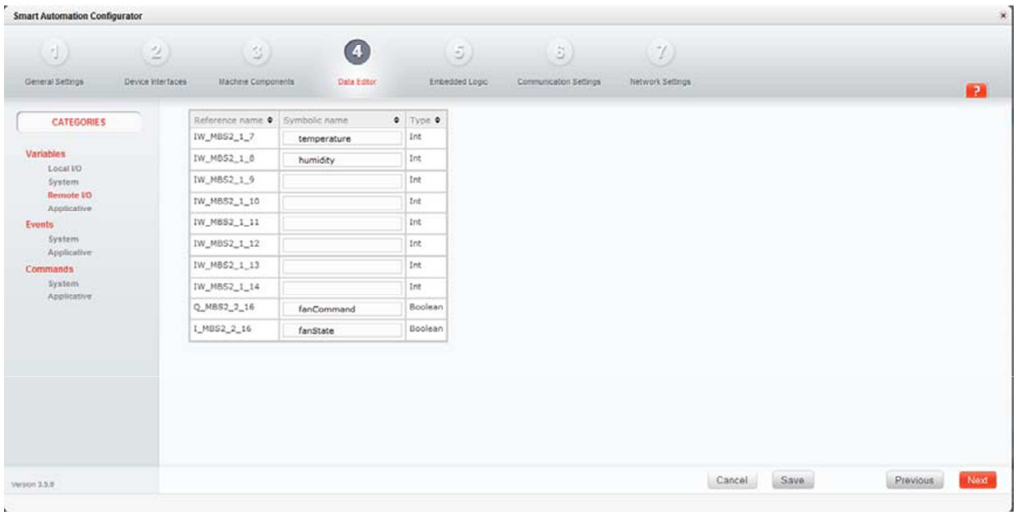
The user application applies a set of rules on the device's I/O.

- Description of rules & config on web forms
- Embedded code generation
- Server config generation
- Automated deployment



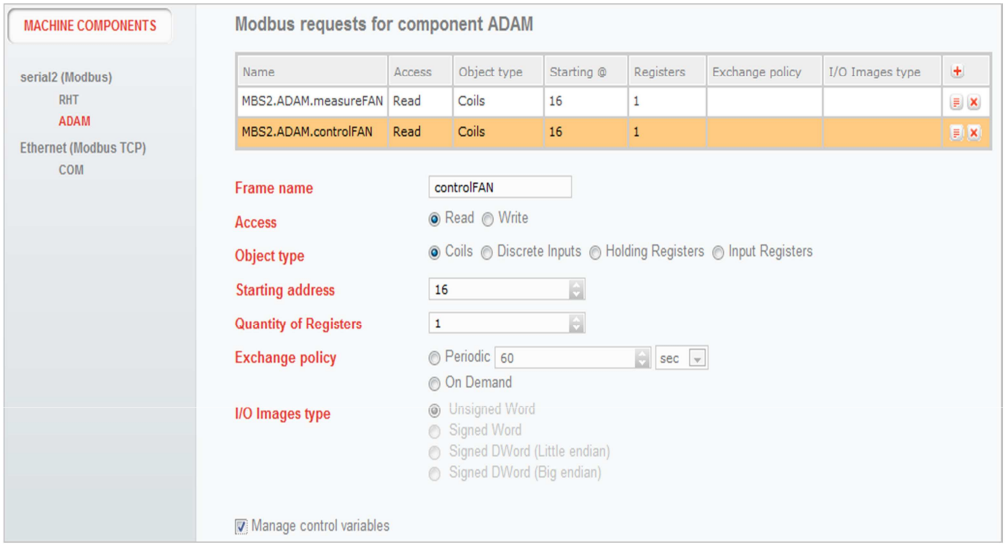
SmartAutomation config

Configure connection policy, I/Os, monitoring rules…



A screenshot from the app definition pages of SmartAutomation (mapping Modbus registers to sane variable names).

SmartAutomation config



A screenshot from the app definition pages of SmartAutomation (describing the monitorin policy on a variable).

Standard front-end

SIERRA WIRELESS

Once the embedded application is deployed, it can be monitored through the usual front-end.

Lua framework services

- **Scheduler** + IPC, synchro, mutexes
- **Logs**: verbosity management, storage, redirection
- Hessian and Lua-friendly **serialization** (much lighter than Pluto, still supports realistic things: functions, shared sub-tables…)
- Architecture-specific **I/O**s: serial, GPIO, modbus…
- **Network** management (IP interfaces)

These services are portable, and not embedded-specific. Should soon be offered under a free and business-friendly license.

Development comfort

Some services ease the embedded-development-specific pains:

- Telnet (+ edition, auto-complete, etc. Can run over UART as well as TCP/IP).
- Interactive loading, reloading of source files over HTTP.
- Configurable logs.
- Can throw new inspection threads in, without breaking the app.
- More stuff coming with Eclipse support (cf. next presentation).

Lua tweaks

Our tweaks against Lua 5.1:

- Imported some uncontroversial futures (e.g. __pairs and __ipairs)
- Embedded-friendly standard patches: emergency GC, LNUM
- Custom patches: execute from flash

Moving to Lua 5.2?

- The main feature for us would be emergency GC, but we've got it.
- C continuations ("*attempt to yield across boundary*" errors) is our 2nd favorite feature. Forced us to rewrite some standard stuff in Lua.
- Not looking forward to redo exotic ports…

How would we have done without Lua?

Abandoning portability isn't an option:

- We cannot give up smaller architectures
- We still want to move on to Linux, expect costs to eventually catch up

So, we would have gone with C:

- Callback-driven C, for small architectures support (unmaintainable)
- On Linux, either an extensive rewrite, or shared callback-driven codebase + porting layer, very hard to extend and maintain
- back to debugging `malloc()` and `strcat()`…

Bottom line: huge additional development times and costs

Supposing that we could have run Python / Ruby:

- Concurrency issues would have been much more of a concern
- Even if they come batteries included, those aren't the right batteries for M2M.