IRFT – T2TRG

Programming the IoT with C++ Actors

Dominik Charousset, Raphael Hiesgen, Thomas C. Schmidt

{dominik.charousset, raphael.hiesgen, t.schmidt}@haw-hamburg.de

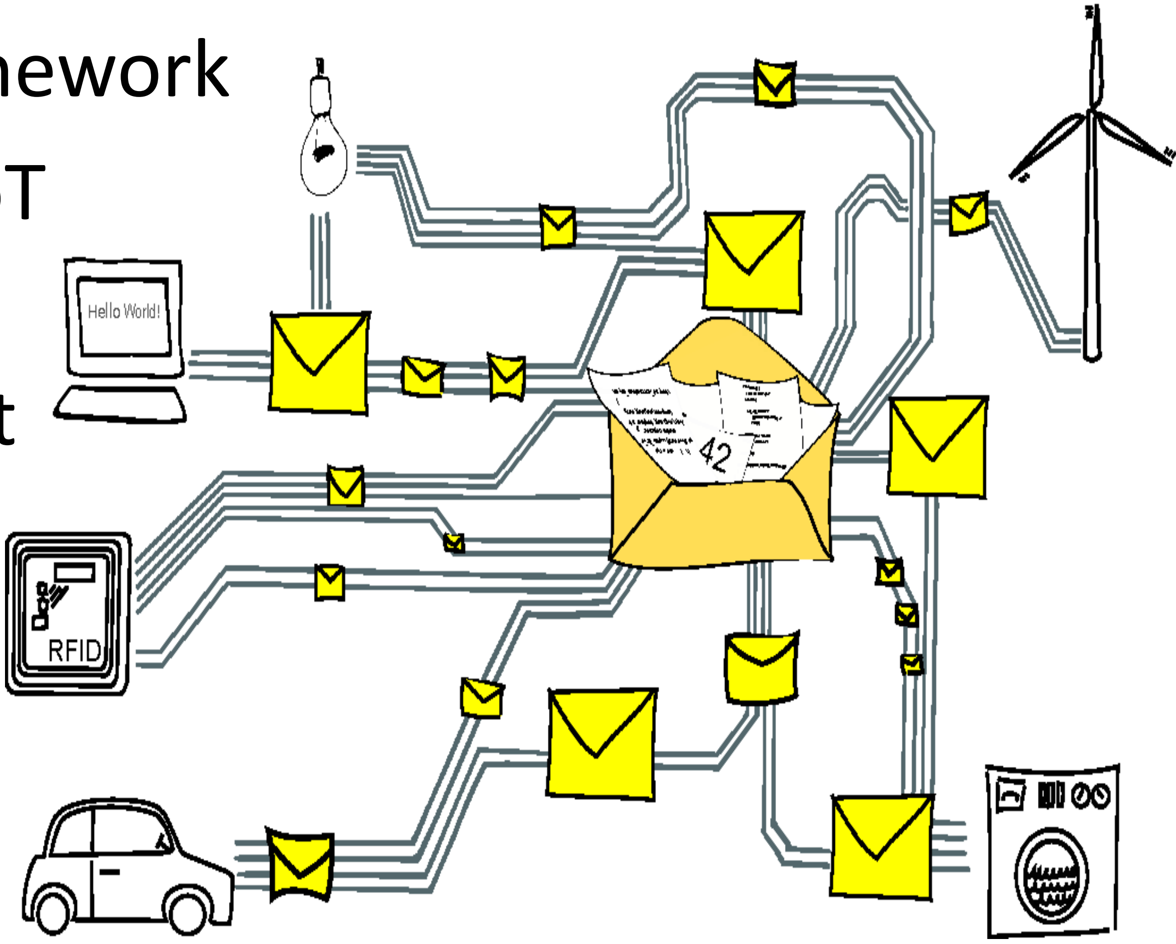
Internet Technologies Group

Hamburg University of Applied Sciences

http://www.haw-hamburg.de/inet

Outline

1. Actor Model
2. C++ Actor Framework
3. Actors in the IoT
4. Use-case
   Intelligent Light



The Actor Model

- Concept by Hewitt, Bishop & Steiger (1973)
- Isolated software entities: actors
- Lightweight concurrency abstraction:
  $\rightarrow$ Actors are sub-thread entities
- Network-transparent message passing
- Divide & conquer via “spawn”
- Strong, hierarchical failure model
- Re-deployment at runtime

C++ Actor Framework - CAF

- Open Source Community since 2011
  - Developers from Europe, North America & Asia
  - https://github.com/actor-framework/
  - ~ 75 Forks on GitHub
- Focus on Scalabiltiy
  - UP: Thousands of Cores
  - Down: IoT Nodes
  - Sideways: GPUs
- Significant User Base
  - In powerful environments
  - Now running at RIOT

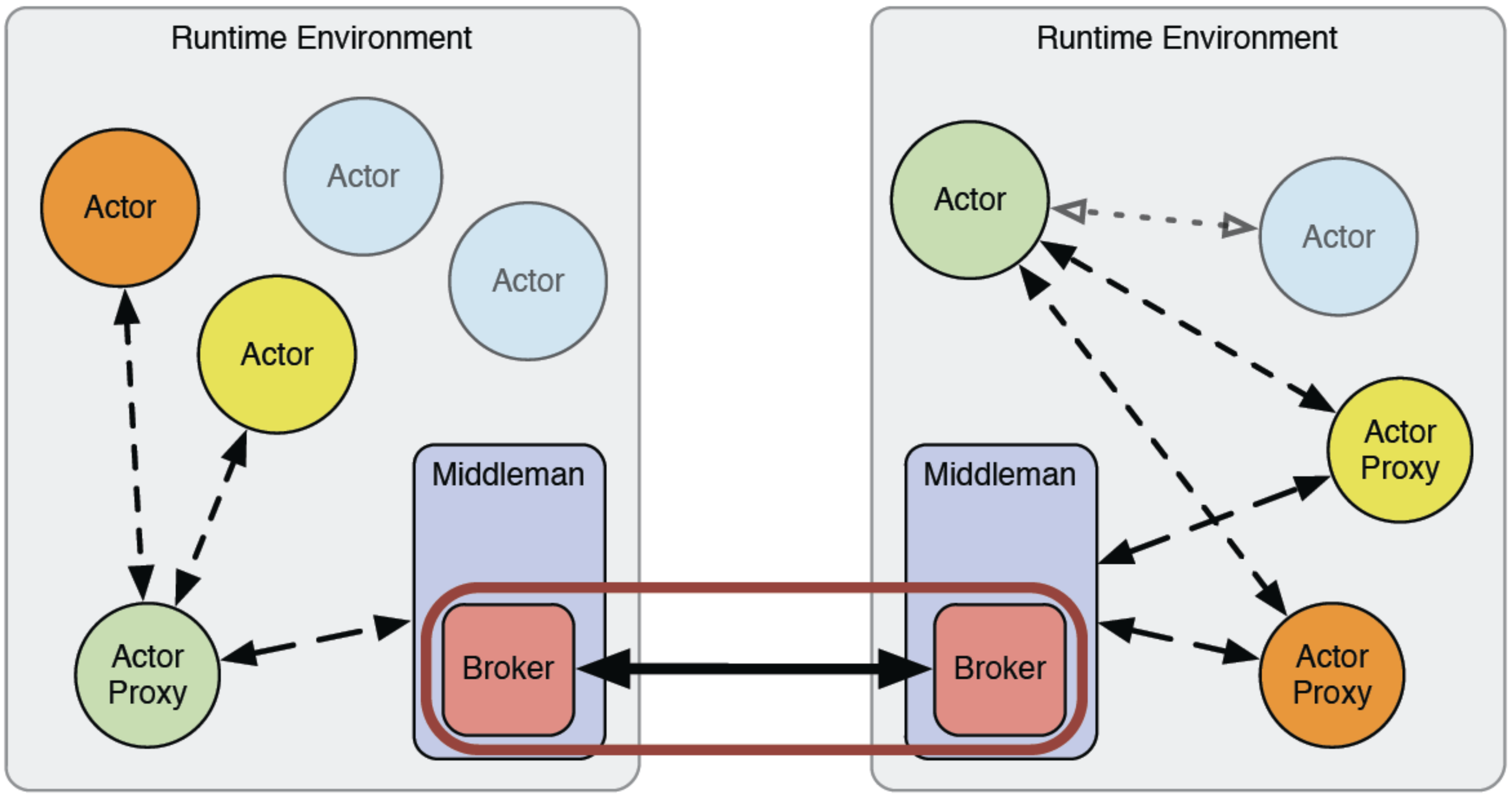
CAF@IoT: Goals and Potential

- Professionalize IoT software development
  - Reusability, robustness and portability
  - “Internet-like” distributed programming
- Establish a common programming model
  - Highly distributed application design
  - Higher level of abstraction
  - Distributed error-handling
- Promote experimentally driven research
  - IoT environments often unpredictable
  - Facilitate controlled software environment

Core Approach

- Network stack based on 6LoWPAN and UDP
- Message handling via CoAP
  - Asynchronous transactions
  - Duplicate message detection
  - Optional reliability
- Security
  - Encryption and integrity: DTLS
  - Authentication: ID-based crypto with ECC

CAF Software Architecture



Communication with CAF

- Detect / correct errors on a distributed actor level
- Types give data meaning
  - Celsius vs. Fahrenheit, Feet vs. Meter, …
  - Compile time validation of message types
- Publish / subscribe pattern for sensors
- Brokers can translate to different data formats
  - Protobuf, CBOR, …
- Group communication eases service discovery and rendezvous processes

Use Case: Intelligent Lightening

Intelligent lightening system switches lights by predicting user demands. This includes third party information from elevators, doors, etc.

- Distributed heterogeneous system
  - Cooperation between lights, elevators, locks, …
- Open Interfaces
  - Messaging standards for application domains
  - Allow cooperation of different vendors, i.e., lights and elevators
- Dynamic error handling
  - Raise maintenance alarm
  - Turn nearby lights on as a fallback

Setup

- Motion Sensor
  - Detect movement in the area
  - Execute callback to notify lights
- Lights
  - Turn on if triggered by another sensor
  - Turn off after timeout
- Elevators & Locks
  - Trigger lights at the destinations of users

Devices Interact Directly

On the Sensor-Light: Idle

```
// self is used to access actor specific functionality
// passed by the runtime as the first argument
behavior idle(event_based_actor* self) {
  return {
    // atoms are annotations to give messages
    // meaning beyond their types
    [=](on_atom) {
      turn_light_on();
      // 1st arg allows us to change back to this behavior
      // 2nd arg is the new behavior, see next slide
      self->become(keep_behavior, glowing(self));
    }
  };
}
```

On the Sensor-Light: Glowing

```
behavior glowing(event_based_actor* self) {
  return {
    // receiving an on_atom resets the timer
    [=](on_atom) { /* NOP */ },
    // turn off if no other message
    // is received within the timeout
    after(TIMEOUT_TIME) >> [] {
      turn_light_off();
      // switch back to last behavior: idle
      self->unbecome();
    }
  };
}
```

Design Space

- Motion sensor
  - High level API to register interested actors:

    ```
    motion_sensor.add_listener(light);
    ```
- Elevator control (multiple elevators)
  - Sensors in the same well register in a group (control panels, supervisors, …)
  - Elevators subscribe to group of control messages
  - Multiple elevators require complex scheduling

On the Elevator: Notify Lights

```
// Handle notifications from elevator control
behavior elevator(event_based_actor* self) {
  return {
    // received when a button is pressed
    [=](request_atom, int floor) {
      register_destination(floor);
    },
    // received before arriving at a new floor
    [=](arriving_atom, int floor) {
      if (has_request_for(floor)) {
        self->send(actor_located_at(floor), on_atom);
      }
    }
  }; }
```