Building Virtual Devices

using the plan44.ch vdcd framework

Lukas Zeller, plan44.ch

2013-11-12

plan44.ch?

- small hard- and software engineering company, self owned
- Started experimenting with dS thanks to digitalSTROM developer days 2011 and help on mailing list
- Motivated by the fact that much of the dS stack is open source
- Experiments evolved into the "plan44 digitalSTROM bridge" project in 2012
- Now using dS open source and contributing to it with the virtual device connector (vdc) implementation presented here

"vDC" - virtual device connector

"vDC" - virtual device connector

- Connects non-dS hardware to a dS system

"vDC" - virtual device connector

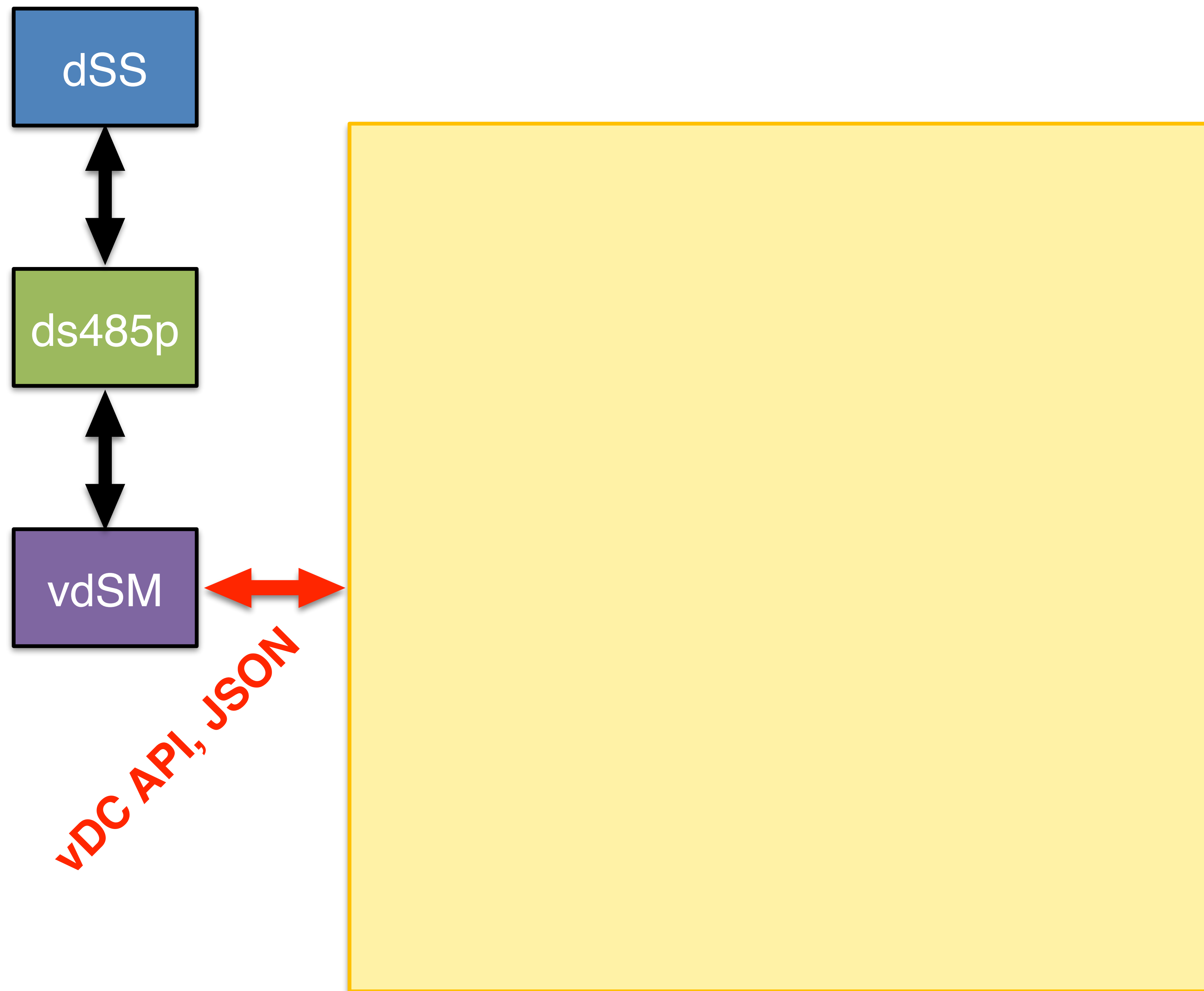
- Connects non-dS hardware to a dS system
- makes external hardware "look" like digitalSTROM-Devices to the system

"vDC" - virtual device connector

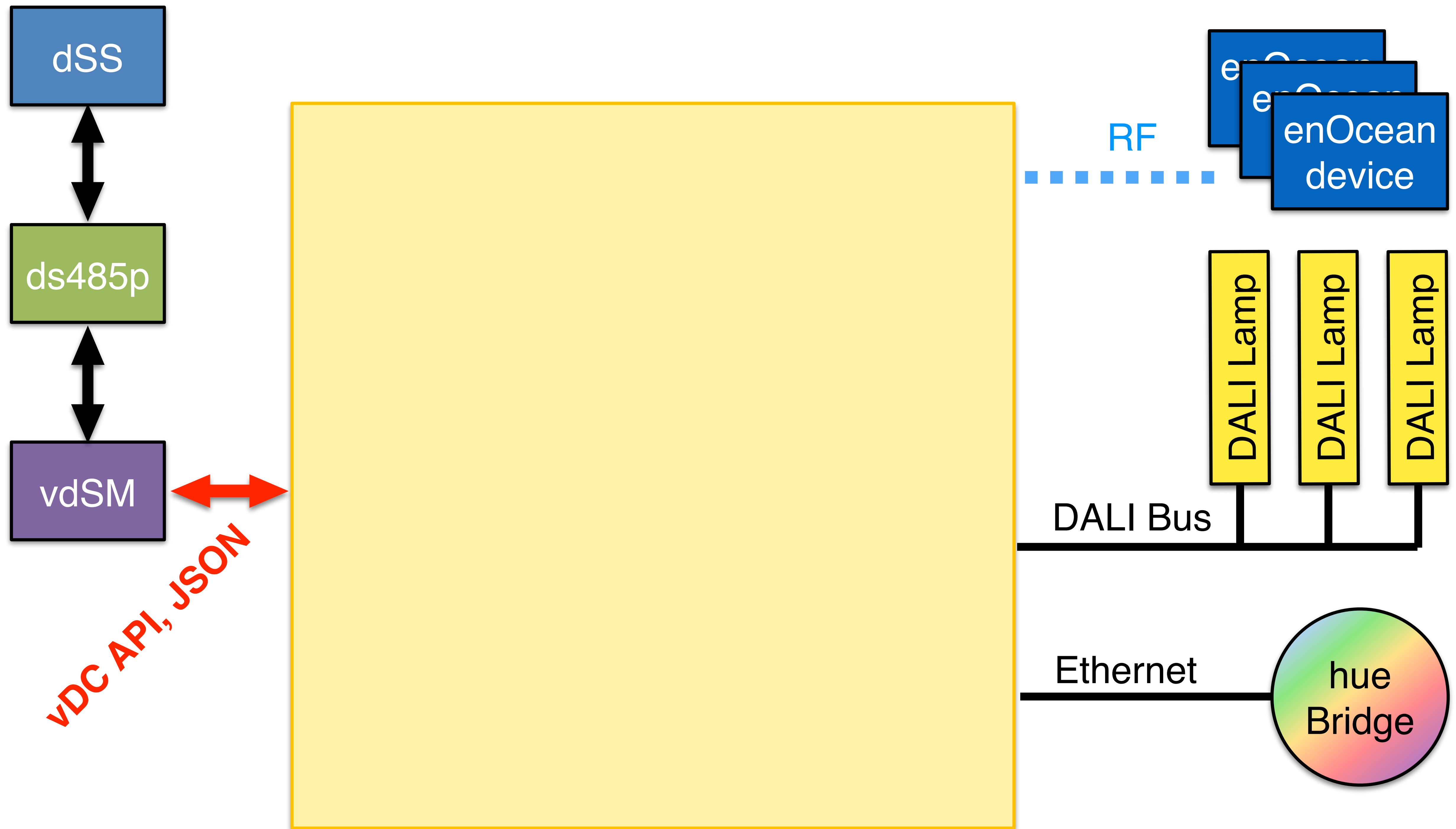
- Connects non-dS hardware to a dS system
- makes external hardware "look" like digitalSTROM-Devices to the system
- does so by wrapping each hardware device into a "virtual device" or "**vdSD"**

vDC - virtual device connector

vDC - virtual device connector

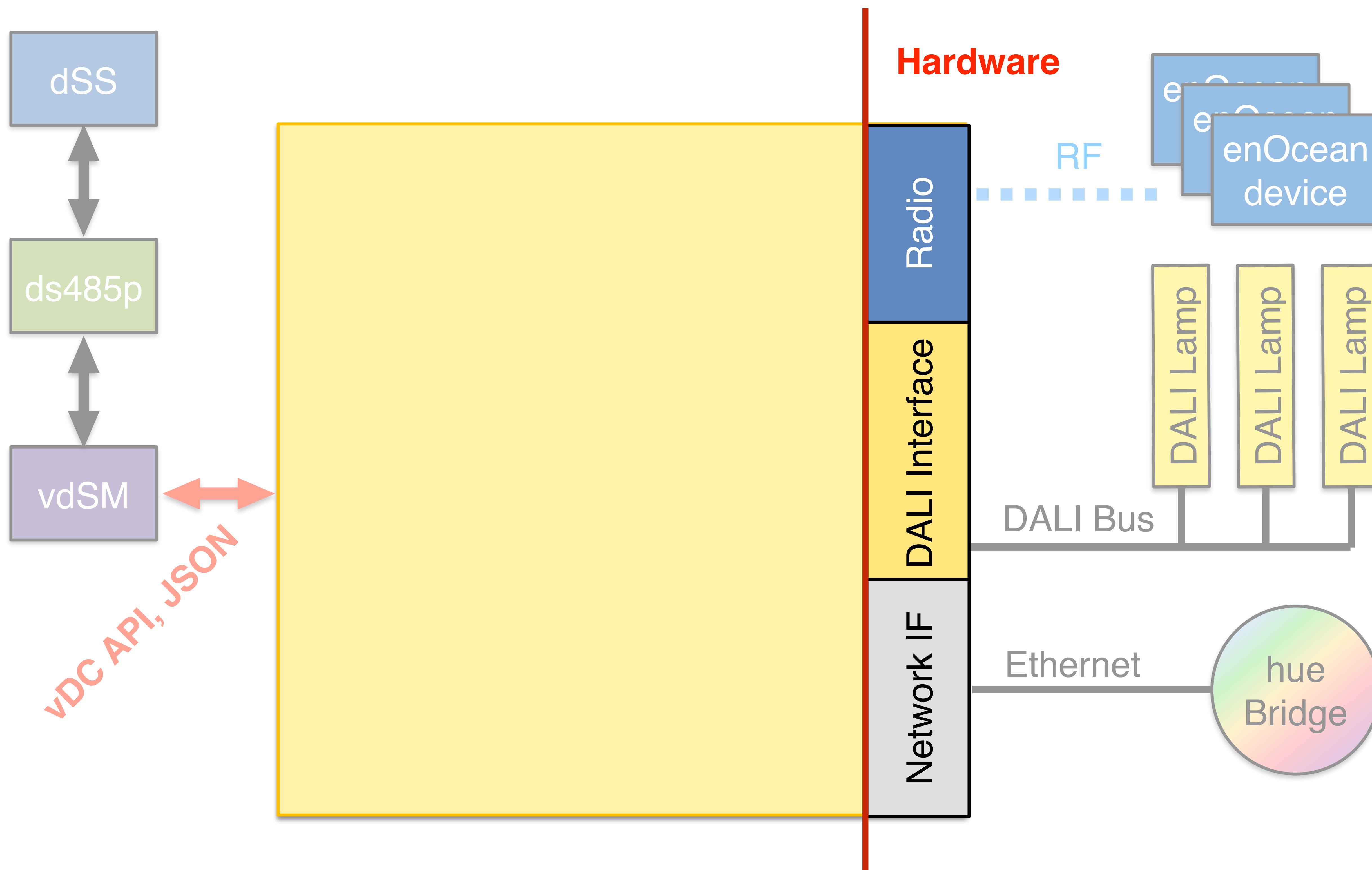


vDC - virtual device connector

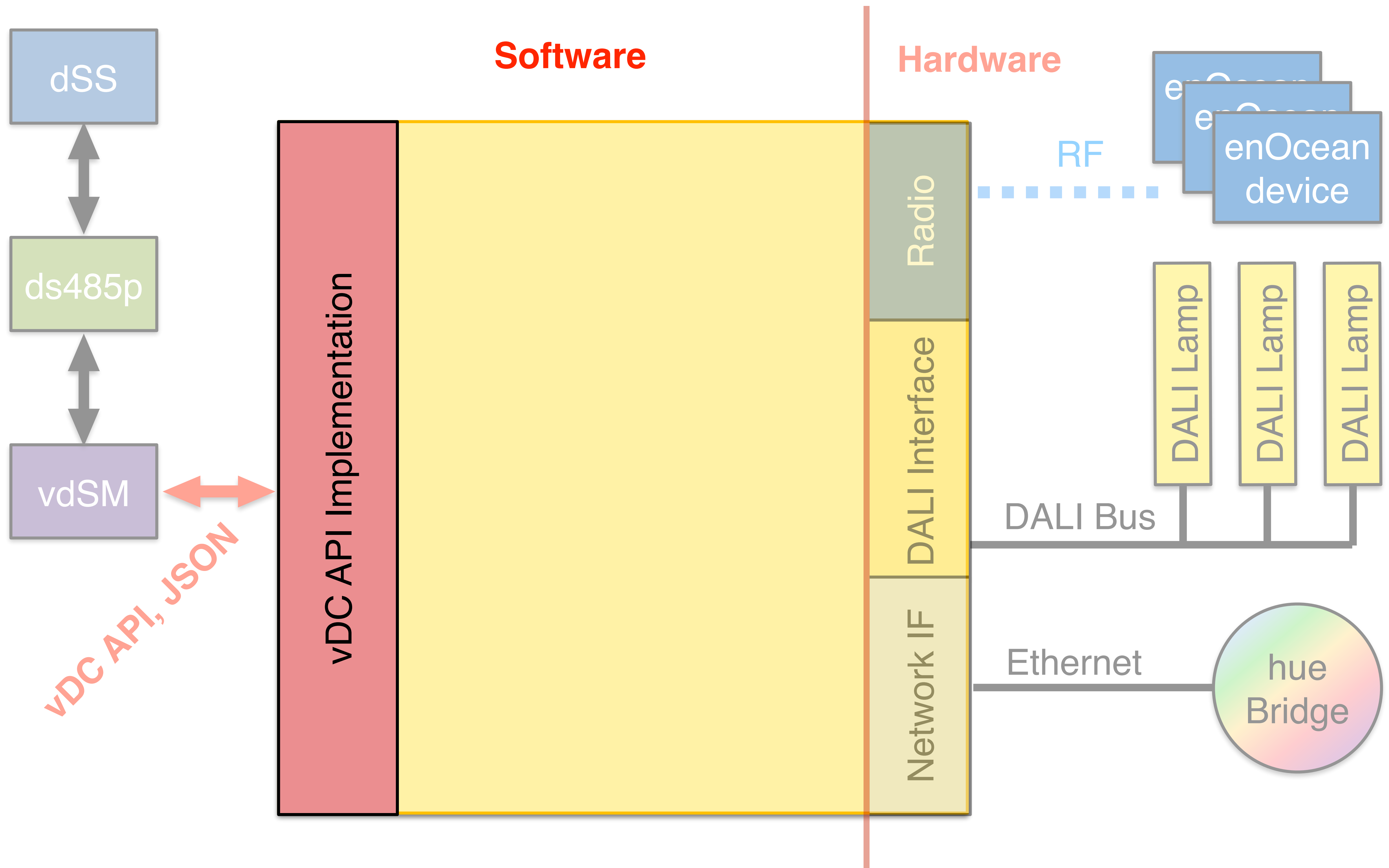


plan44.ch

vDC - virtual device connector

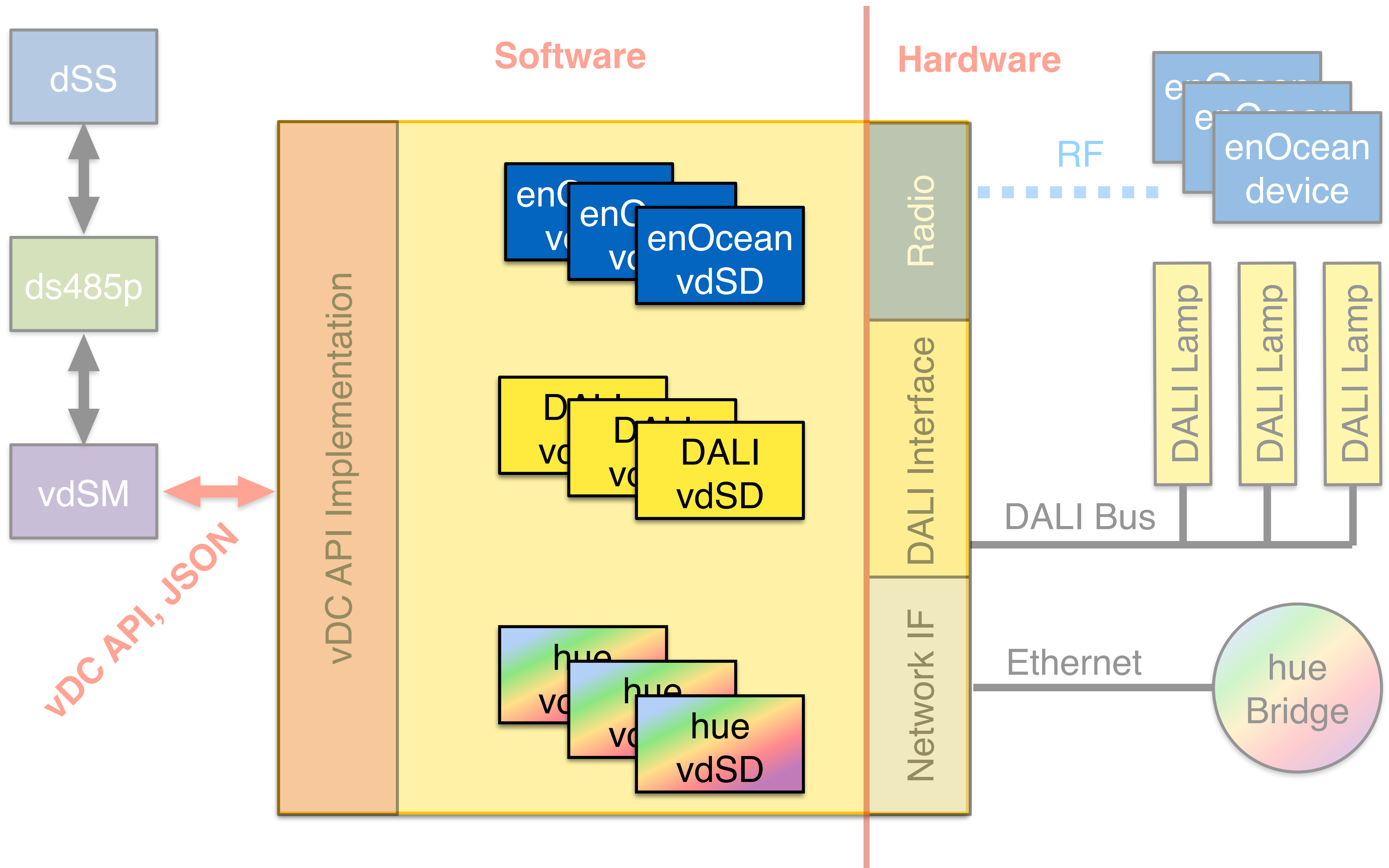


vDC - virtual device connector



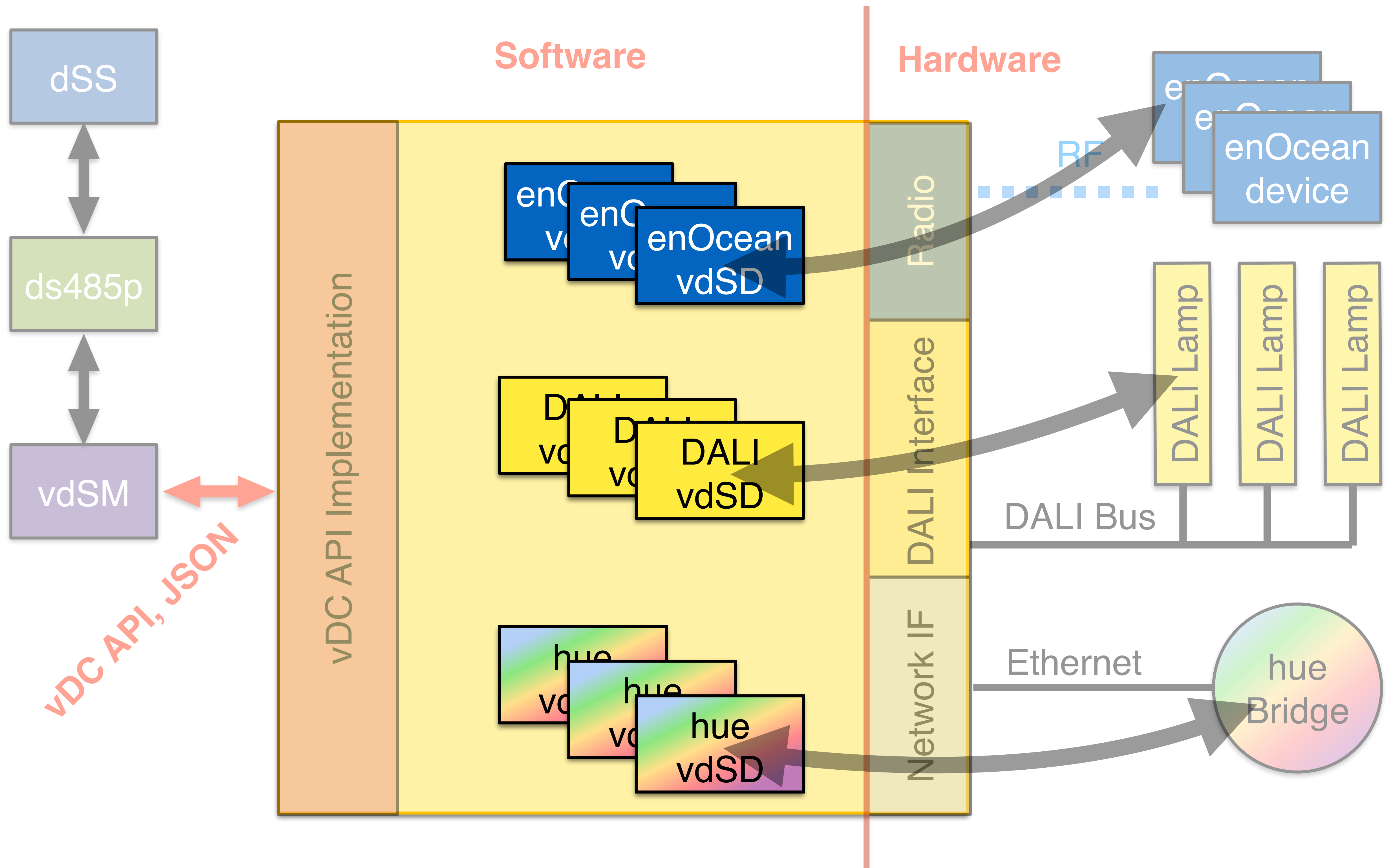
plan44.ch

vDC - virtual device connector



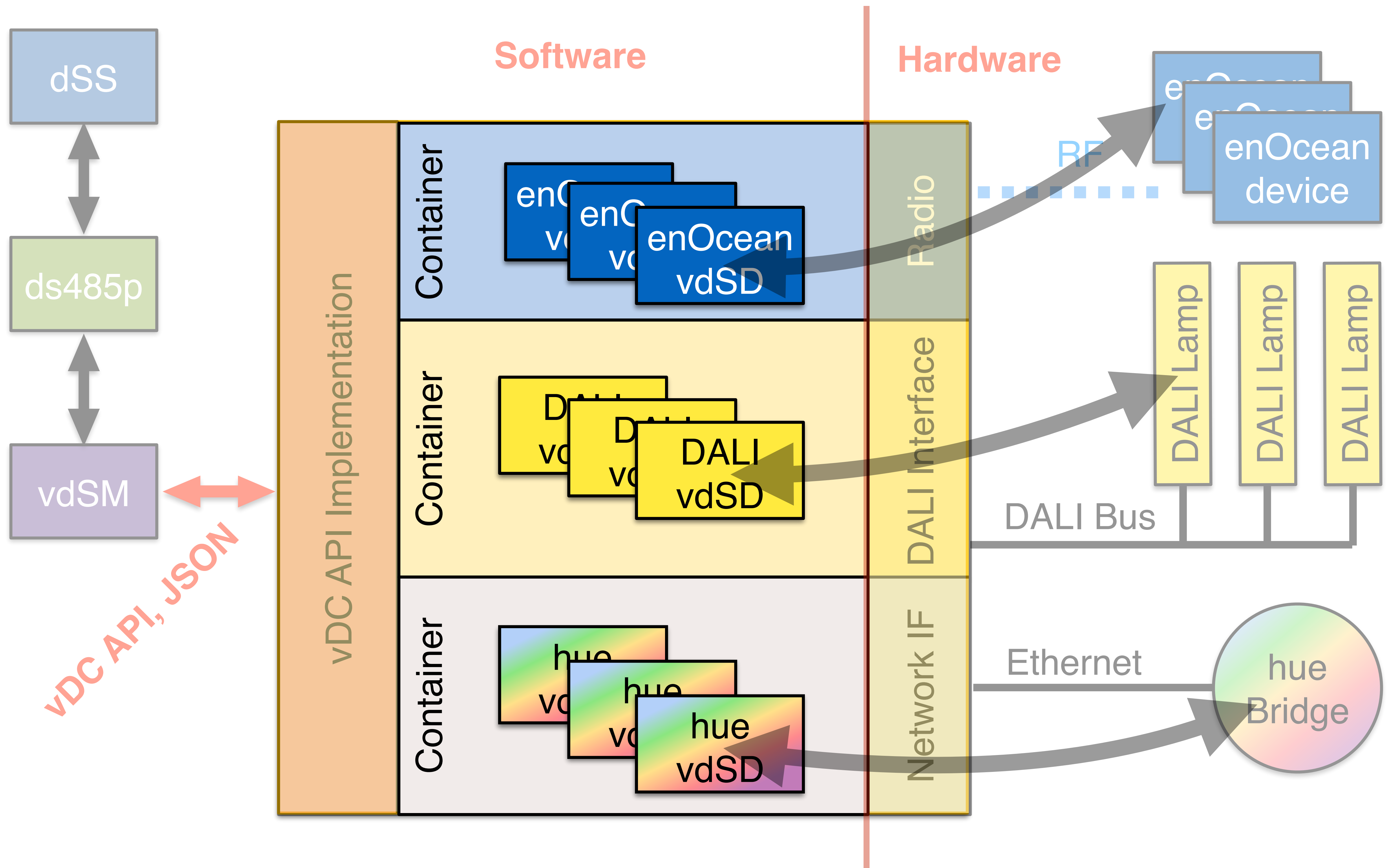
plan44.ch

vDC - virtual device connector



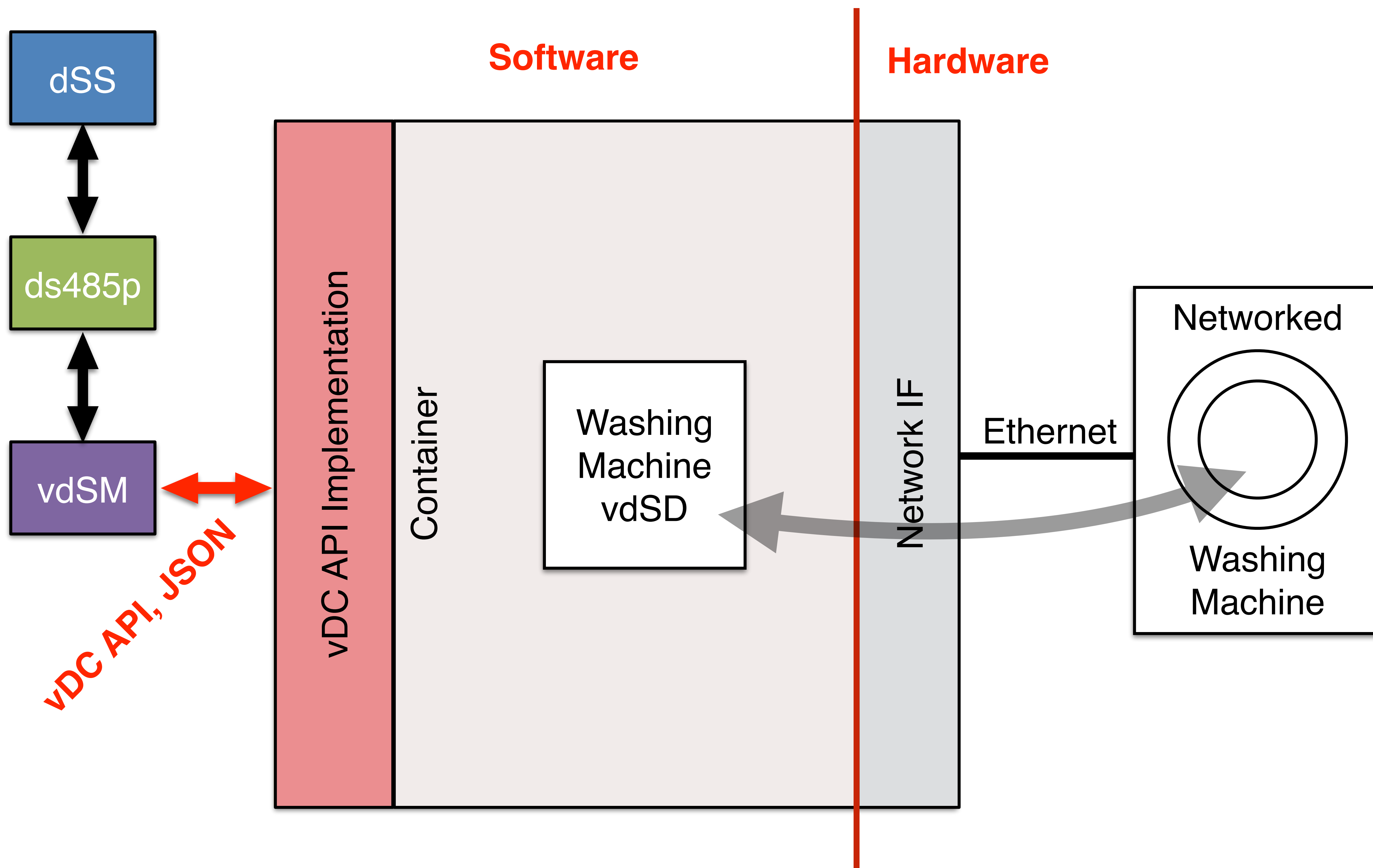
plan44.ch

vDC - virtual device connector



plan44.ch

simple vDC for single device



vdSD - virtual device

- "talks" with the actual device in its native language (protocol)
- "translates" properties, events and values into the dS logic
- implements standard dS device behaviours like
  - dimming and scenes for lights
  - click detection for buttons
  - value reporting for sensors
- describes the device structure (how many inputs/outputs etc.)
- "talks" to the dS system via vDC API

vdSD - virtual device

- "talks" with the actual device in its native language (protocol)
- "translates" properties, events and values into the dS logic

Specific

- implements standard dS device behaviours like
  - dimming and scenes for lights
  - click detection for buttons
  - value reporting for sensors
- describes the device structure (how many inputs/outputs etc.)
- "talks" to the dS system via vDC API

vdSD - virtual device

- "talks" with the actual device in its native language (protocol)
- "translates" properties, events and values into the dS logic

Specific

- implements standard dS device behaviours like
  - dimming and scenes for lights
  - click detection for buttons
  - value reporting for sensors
- describes the device structure (how many inputs/outputs etc.)
- "talks" to the dS system via vDC API

Same for all devices

vdSD - virtual device

- "talks" with the actual de [illegible] tive language (protocol)
- "translates" properties, ev [illegible] es into the dS logic

You!

Specific

- implements standard dS device behaviours like
  - dimming and scenes for lights
  - click detection for buttons
  - value reporting for sensors
- describes the device structure (how many inputs/outputs etc.)
- "talks" to the dS system via vDC API

Same for all devices

plan44.ch

vdSD - virtual device

- "talks" with the actual dev... ...tive language (protocol)
- "translates" properties, ev... ...es into the dS logic
- implements standard dS device behaviours like
  - dimming and scenes for lights
  - click detection for bu...
  - value reporting for s...
- describes the device stru... (...ow many inputs/outputs etc.)
- "talks" to the dS system via vDC API

Specific

Same for all devices

You!

The vDC framework

to build a light vdSD…

to build a light vdSD…

- Create a subclass of **`Device`**, configure it as yellow
  You get for free: vDC API

to build a light vdSD…

- Create a subclass of **`Device`**, configure it as yellow
  You get for free: vDC API

- Add a **`LightDeviceSettings`** object
  You get for free: all standard properties a dS light needs, including a persistent scene table

to build a light vdSD…

- Create a subclass of **`Device`**, configure it as yellow
  You get for free: vDC API

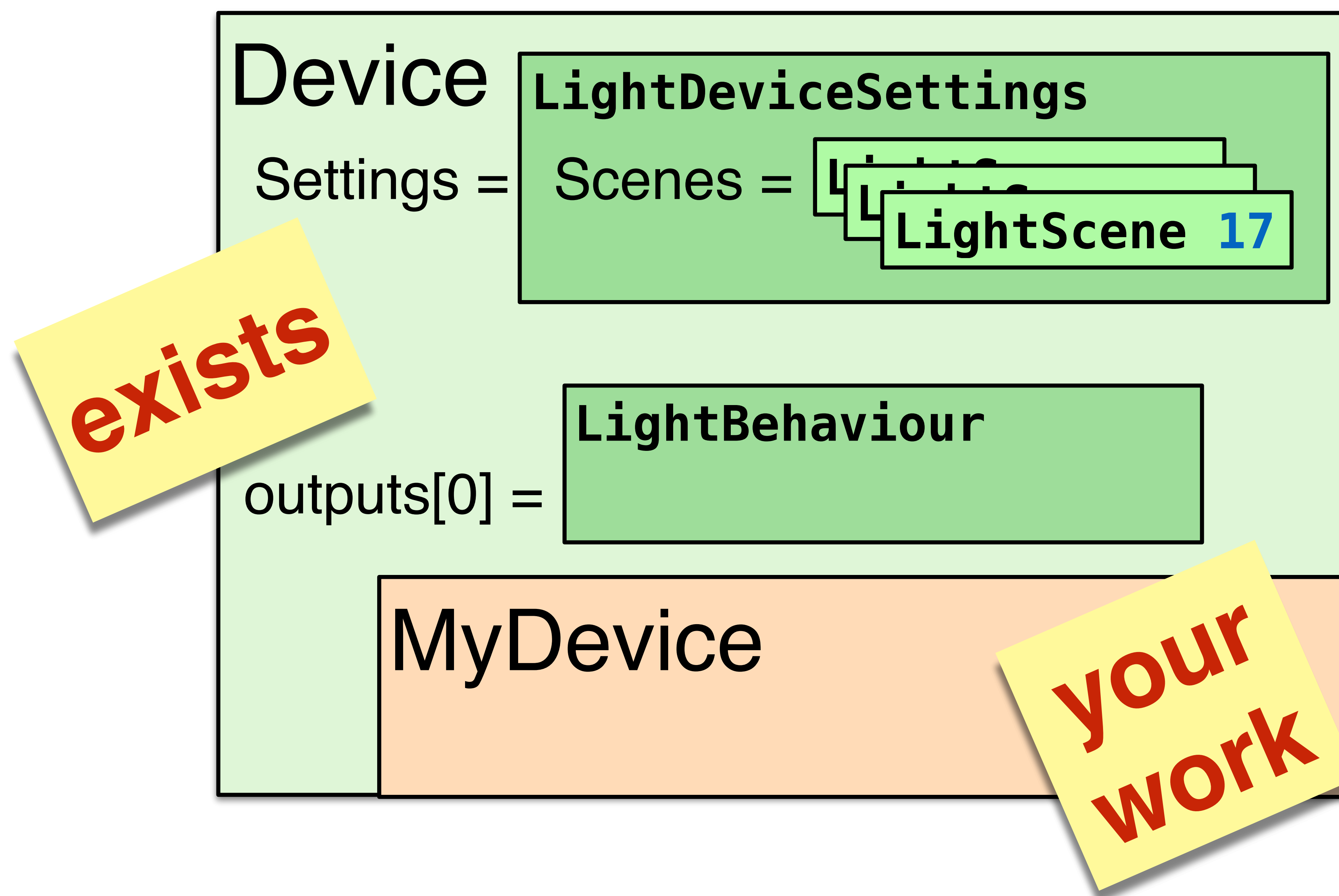
- Add a **`LightDeviceSettings`** object
  You get for free: all standard properties a dS light needs, including a persistent scene table

- Add one **`LightBehaviour`** object, configure it to be a dimmer
  You get for free: standard dS dimming behaviour, calling and saving scenes

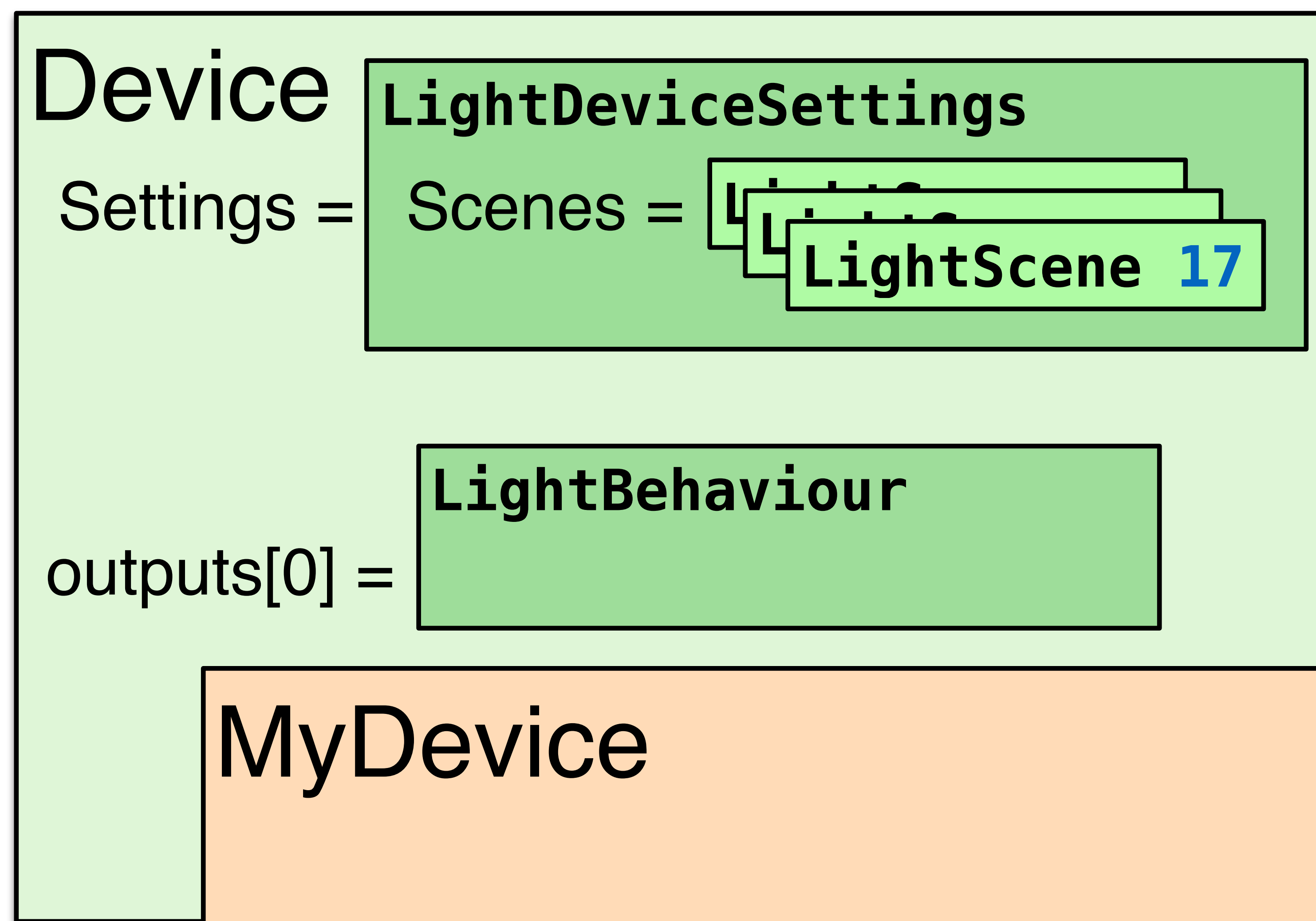
to build a light vdSD...

- Create a subclass of **`Device`**, configure it as yellow
  You get for free: vDC API
- Add a **`LightDeviceSettings`** object
  You get for free: all standard properties a dS light needs, including a persistent scene table
- Add one **`LightBehaviour`** object, configure it to be a dimmer
  You get for free: standard dS dimming behaviour, calling and saving scenes
- Now implement one method, **`updateOutputValue()`**
  Nothing free here - that's your task :-)

virtual light device operation

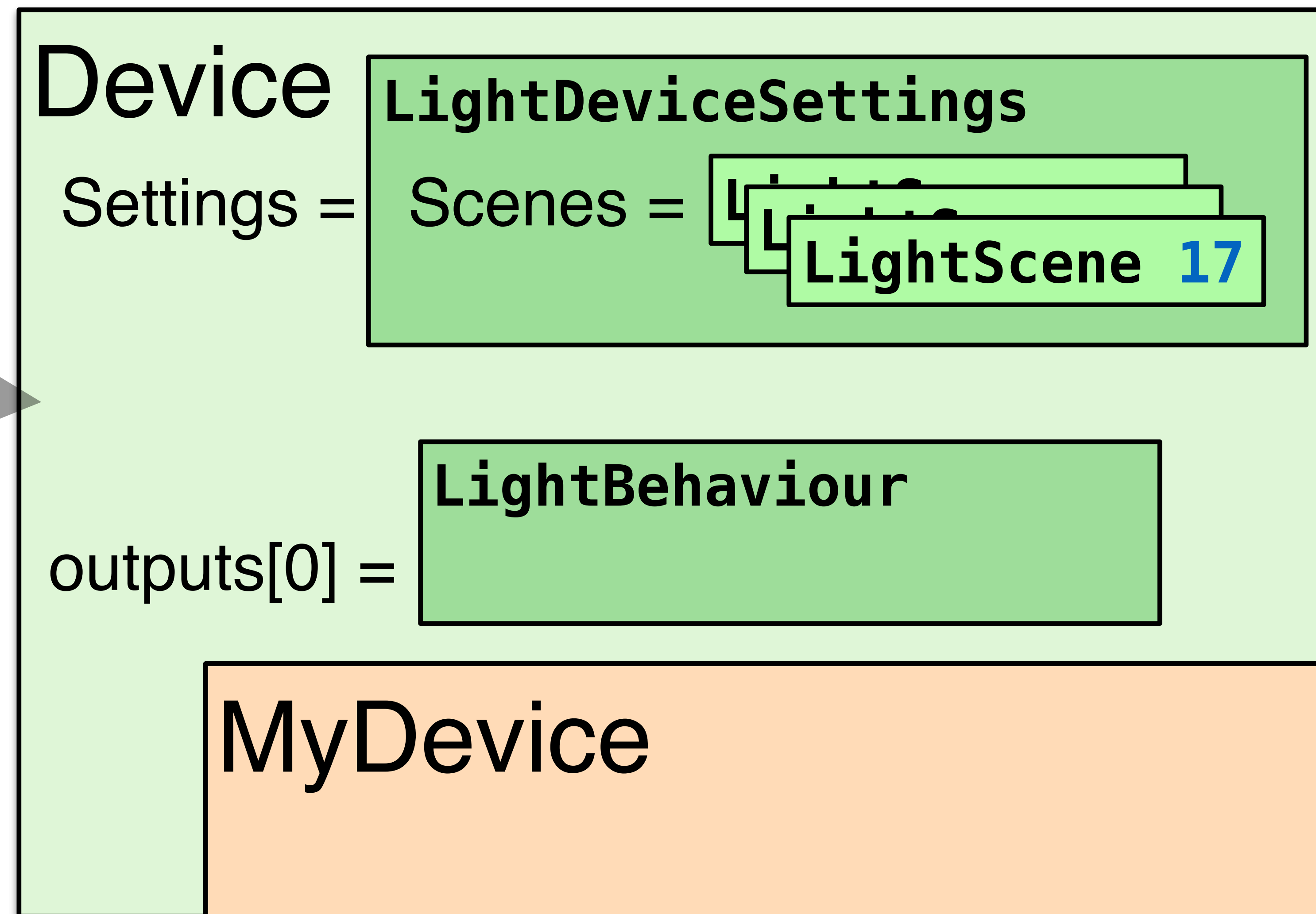


virtual light device operation



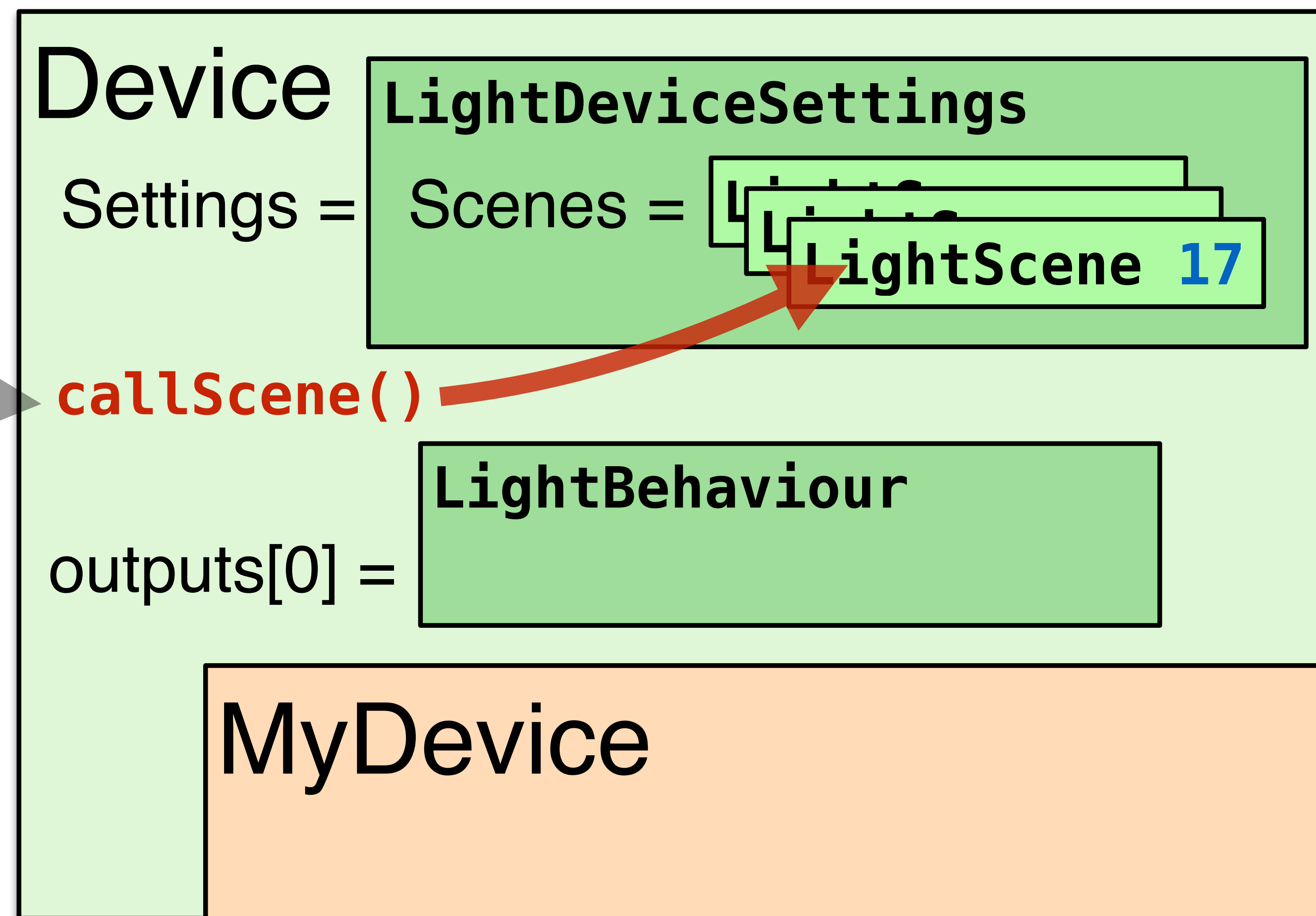
virtual light device operation

```
{
   "jsonrpc":"2.0",
   "method":"callScene",
   "params":{
      "dSUID":"3504175FE0000000090656D0",
      "scene":17,
      "force":false
   }
}
```



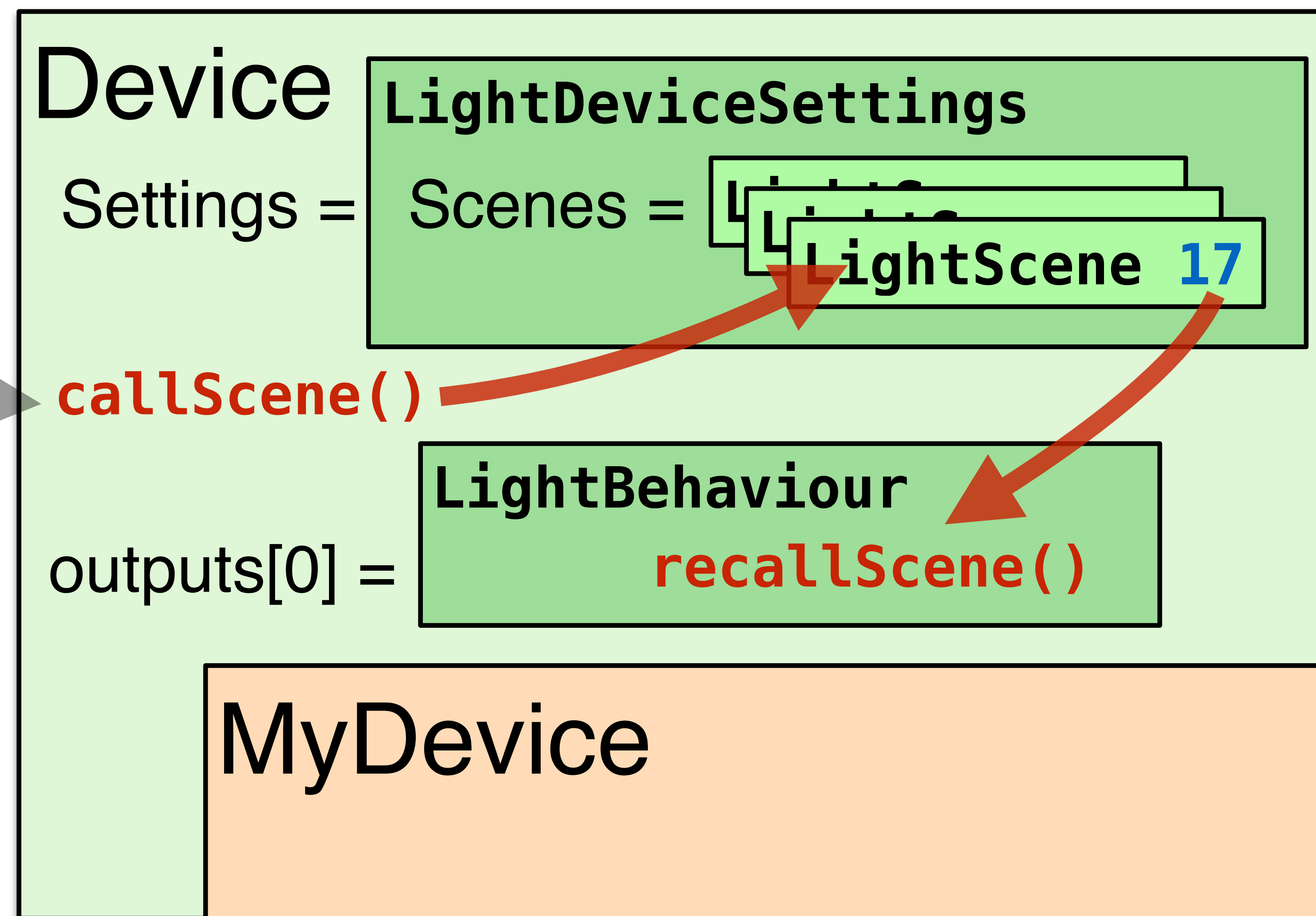
virtual light device operation

```
{
   "jsonrpc":"2.0",
   "method":"callScene",
   "params":{
      "dSUID":"3504175FE0000000090656D0",
      "scene":17,
      "force":false
   }
}
```



virtual light device operation

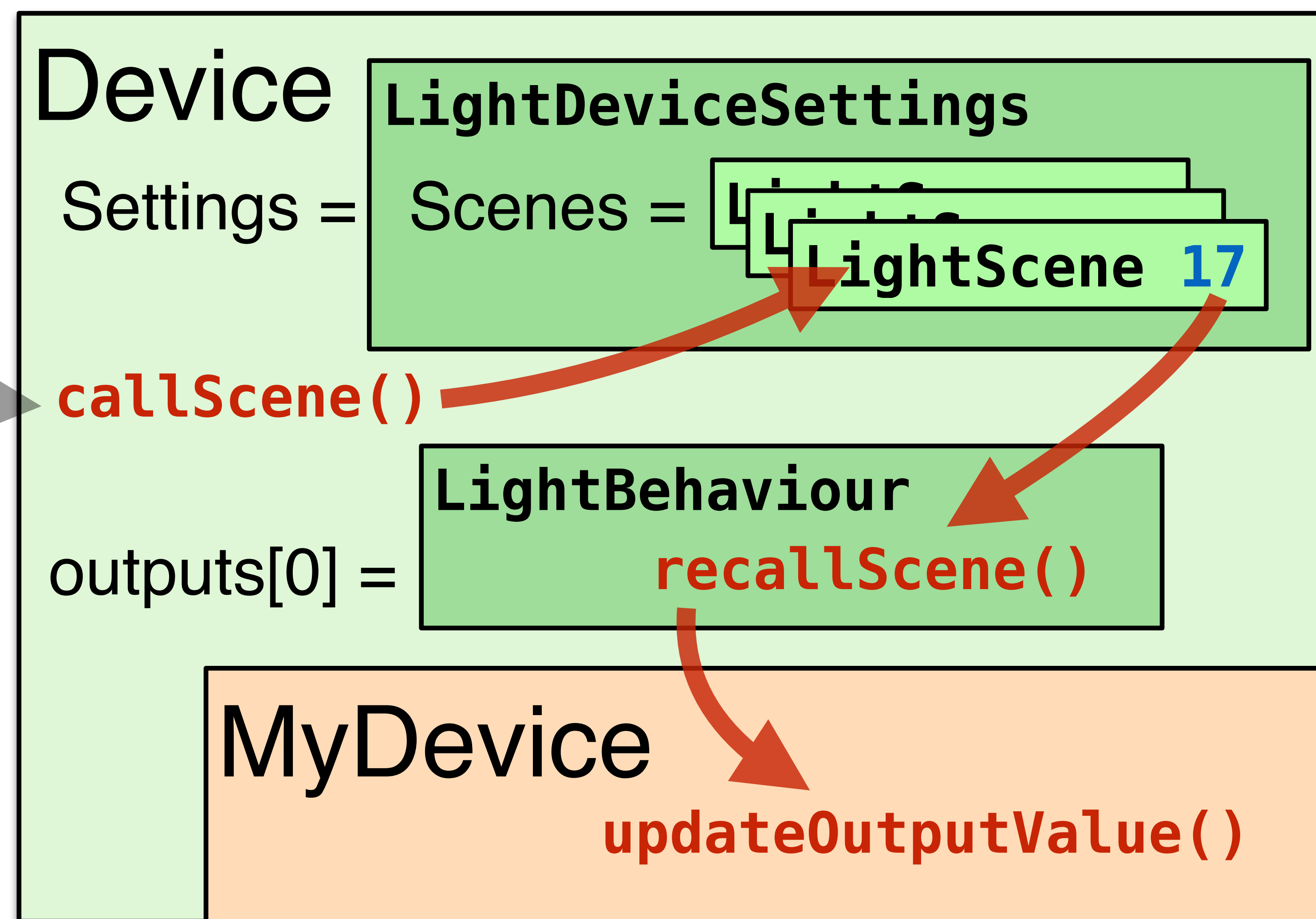
```
{
   "jsonrpc":"2.0",
   "method":"callScene",
   "params":{
      "dSUID":"3504175FE0000000090656D0",
      "scene":17,
      "force":false
   }
}
```



plan44.ch

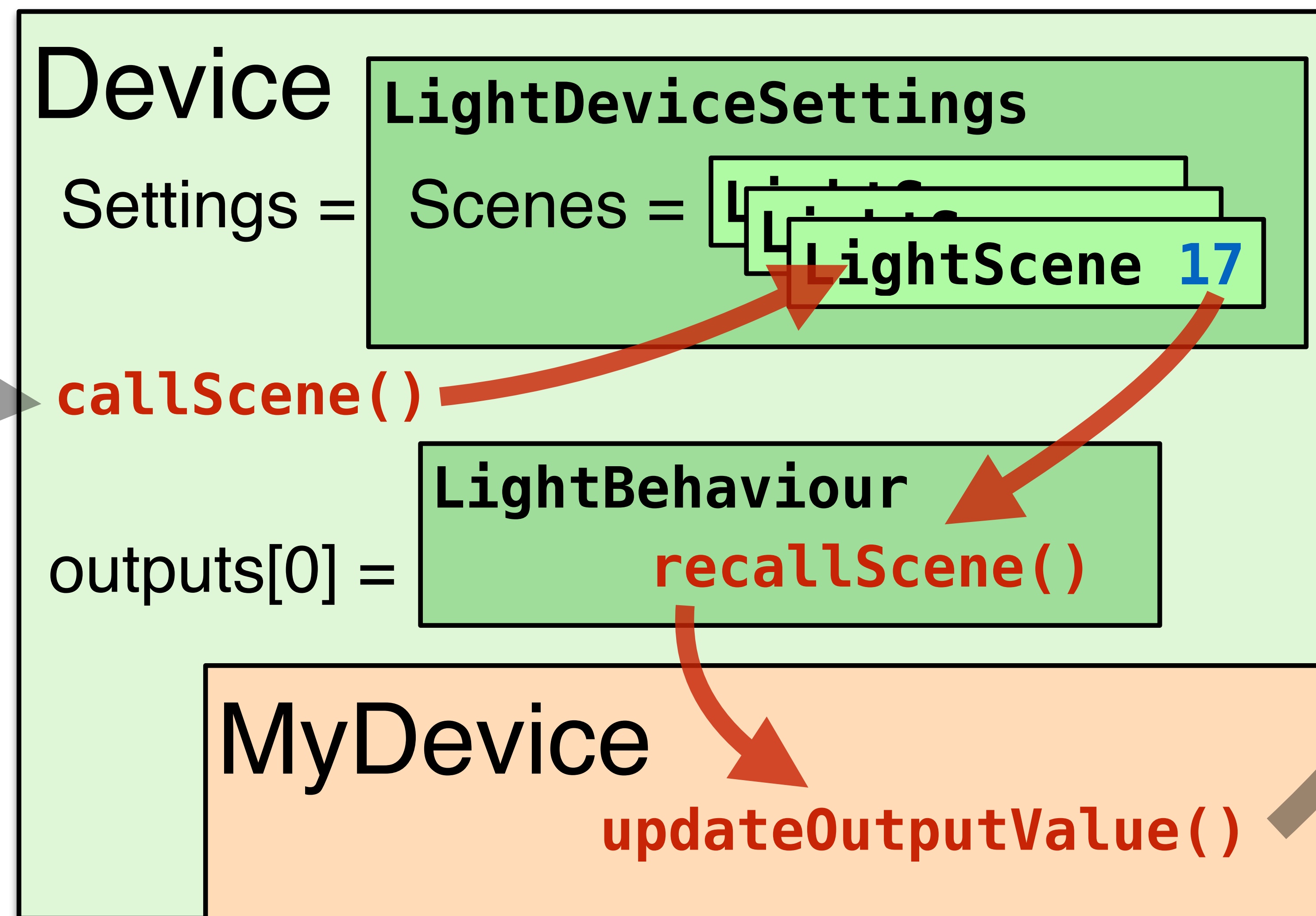
virtual light device operation

```
{
   "jsonrpc":"2.0",
   "method":"callScene",
   "params":{
      "dSUID":"3504175FE0000000090656D0",
      "scene":17,
      "force":false
   }
}
```



virtual light device operation

```
{
   "jsonrpc":"2.0",
   "method":"callScene",
   "params":{
      "dSUID":"3504175FE0000000090656D0",
      "scene":17,
      "force":false
   }
}
```



What's left to do?

What's left to do?

- **`deriveDsUid()`**:
  - Each device must have an unique dSUID. vDC helps to derive this from hardware identifiers such as MAC addresses.

What's left to do?

- **`deriveDsUid()`**:
  - Each device must have an unique dSUID. vDC helps to derive this from hardware identifiers such as MAC addresses.
- **`MyClassController::collectDevices()`**:
  - if your devices are on a bus, you need to scan the bus and instantiate a virtual device for each hardware device found.
  - for a dedicated vdc (washing machine), collectDevices() just instantiates one single virtual device on startup.

The vdc toolbox

- C++, light use of boost (smart pointers and binding), compiles with gcc down to 4.3
- based on mainloop with timed and I/O event related callbacks
- works single-threaded (but you can use threads if you need to)
- SQLite3 based persistence, abstracted with mostly automatic schema update
- Standard behaviours for generic outputs, light, pushbuttons, binary inputs and sensors - more to come as dS evolves.
- Extendable scene table (e.g. color in addition to brightness)
- Utilities for serial communication, sockets, web services, SSDP, GPIOs, i2c based I/O etc.

```
DemoDevice::DemoDevice(DemoDeviceContainer *aClassContainerP) :
  Device((DeviceClassContainer *)aClassContainerP)
{
  // a demo device is a light which shows its dimming value as a string of 0..50 hashes on the console
  // - is a light device
  primaryGroup = group_yellow_light;
  // - use light settings, which include a fully functional scene table
  deviceSettings = DeviceSettingsPtr(new LightDeviceSettings(*this));
  // - create one output with light behaviour
  LightBehaviourPtr l = LightBehaviourPtr(new LightBehaviour(*this));
  // - set default config to act as dimmer with variable ramps
  l->setHardwareOutputConfig(outputFunction_dimmer, usage_undefined, true, -1);
  addBehaviour(l);
  // - hardware is defined, now derive dSUID
 deriveDsUid();
}


void DemoDevice::updateOutputValue(OutputBehaviour &aOutputBehaviour)
{
  // as this demo device has only one output
  if (aOutputBehaviour.getIndex()==0) {
    // This would be the place to implement sending the output value to the hardware
    // For the demo device, we show the output as a bar of 0..50 '#' chars
    // - read the output value from the behaviour
    int hwValue = aOutputBehaviour.valueForHardware();
    // - display as a bar of hash chars
    string bar;
    while (hwValue>0) {
      // one hash character per 4 output value steps (0..255 = 0..64 hashes)
      bar += '#';
      hwValue -= 4;
    }
    printf("Demo Device Output: %s\n", bar.c_str());
  }
  else
    return inherited::updateOutputValue(aOutputBehaviour); // let superclass handle this
}
```

```
DemoDevice::DemoDevice(DemoDeviceContainer *aClassContainerP) :
  Device((DeviceClassContainer *)aClassContainerP)
{
  // a demo device is a light which shows its dimming value as a string of 0..50 hashes on the console
  // - is a light device
  primaryGroup = group_yellow_light;
  // - use light settings, which include a fully functional scene table
  deviceSettings = DeviceSettingsPtr(new LightDeviceSettings(*this));
  // - create one output with light behaviour
  LightBehaviourPtr l = LightBehaviourPtr(new LightBehaviour(*this));
  // - set default config to act as dimmer with variable ramps
  l->setHardwareOutputConfig(outputFunction_dimmer, usage_undefined, true, -1);
  addBehaviour(l);
  // - hardware is defined, now derive dSUID
 deriveDsUid();
}


void DemoDevice::updateOutputValue(OutputBehaviour &aOutputBehaviour)
{
  // as this demo device has only one output
  if (aOutputBehaviour.getIndex()==0) {
    // This would be the place to implement sending the output value to the hardware
    // For the demo device, we show the output as a bar of 0..50 '#' chars
    // - read the output value from the behaviour
    int hwValue = aOutputBehaviour.valueForHardware();
    // - display as a bar of hash chars
    string bar;
    while (hwValue>0) {
      // one hash character per 4 output value steps (0..255 = 0..64 hashes)
      bar += '#';
      hwValue -= 4;
    }
    printf("Demo Device Output: %s\n", bar.c_str());
  }
  else
    return inherited::updateOutputValue(aOutputBehaviour); // let superclass handle this
}
```

```
DemoDevice::DemoDevice(DemoDeviceContainer *aClassContainerP) :
  Device((DeviceClassContainer *)aClassContainerP)
{
  // a demo device is a light which shows its dimming value as a string of 0..50 hashes on the console
  // - is a light device
  primaryGroup = group_yellow_light;
  // - use light settings, which include a fully functional scene table
  deviceSettings = DeviceSettingsPtr(new LightDeviceSettings(*this));
  // - create one output with light behaviour
  LightBehaviourPtr l = LightBehaviourPtr(new LightBehaviour(*this));
  // - set default config to act as dimmer with variable ramps
  l->setHardwareOutputConfig(outputFunction_dimmer, usage_undefined, true, -1);
  addBehaviour(l);
  // - hardware is defined, now derive dSUID
 deriveDsUid();
}


void DemoDevice::updateOutputValue(OutputBehaviour &aOutputBehaviour)
{
  // as this demo device has only one output
  if (aOutputBehaviour.getIndex()==0) {
    // This would be the place to implement sending the output value to the hardware
    // For the demo device, we show the output as a bar of 0..50 '#' chars
    // - read the output value from the behaviour
    int hwValue = aOutputBehaviour.valueForHardware();
    // - display as a bar of hash chars
    string bar;
    while (hwValue>0) {
      // one hash character per 4 output value steps (0..255 = 0..64 hashes)
      bar += '#';
      hwValue -= 4;
    }
    printf("Demo Device Output: %s\n", bar.c_str());
  }
  else
    return inherited::updateOutputValue(aOutputBehaviour); // let superclass handle this
}
```

```
// device class name
const char *DemoDeviceContainer::deviceClassIdentifier() const
{
  return "Demo_Device_Container";
}


/// collect devices from this device class
void DemoDeviceContainer::collectDevices(CompletedCB aCompletedCB, bool aIncremental, bool aExhaustive)
{
  // incrementally collecting Demo devices makes no sense, they are statically created at startup
  if (!aIncremental) {
    // non-incremental, re-collect all devices
    removeDevices(false);
    // create one single demo device
    DevicePtr newDev = DevicePtr(new DemoDevice(this));
    // add to container
    addDevice(newDev);
  }
  // assume ok
  aCompletedCB(ErrorPtr());
}
```

```
// device class name
const char *DemoDeviceContainer::deviceClassIdentifier() const
{
  return "Demo_Device_Container";
}


/// collect devices from this device class
void DemoDeviceContainer::collectDevices(CompletedCB aCompletedCB, bool aIncremental, bool aExhaustive)
{
  // incrementally collecting Demo devices makes no sense, they are statically created at startup
  if (!aIncremental) {
    // non-incremental, re-collect all devices
    removeDevices(false);
    // create one single demo device
    DevicePtr newDev = DevicePtr(new DemoDevice(this));
    // add to container
    addDevice(newDev);
  }
  // assume ok
  aCompletedCB(ErrorPtr());
}
```

```
// device class name
const char *DemoDeviceContainer::deviceClassIdentifier() const
{
  return "Demo_Device_Container";
}


/// collect devices from this device class
void DemoDeviceContainer::collectDevices(CompletedCB aCompletedCB, bool aIncremental, bool aExhaustive)
{
  // incrementally collecting Demo devices makes no sense, they are statically created at startup
  if (!aIncremental) {
    // non-incremental, re-collect all devices
    removeDevices(false);
    // create one single demo device
    DevicePtr newDev = DevicePtr(new DemoDevice(this));
    // add to container
    addDevice(newDev);
  }
  // assume ok
  aCompletedCB(ErrorPtr());
}
```

Thank you for your attention!