DIY Zoning: Protocols

Table of contents

1. Introduction

This section will only cover protocols used by DZ to provide communications between its own modules. While DZ is (or will be shortly) supporting other protocols (Rendezvous, xAP, xPL), they are covered on their own pages.

2. Common Protocol Features

All the protocols used in DZ will have common features:

- Secure transmission support. Currently, either SSLv3 or TLS is used;
- Autoconfiguration ability, or Plug-and-Play (PnP);
- Capabilities discovery (related to PnP, but available even without it).

> **Note:**
> It seems that the protocol features will be treated exactly like micro components: for example, simplifying the DAC2CORE protocol to just providing the data stream will allow to make the DAC itself significantly smaller (OWFS, a one-line Shell script and couple of pages of Perl code), shifting the arrival/departure control responsibilities to CORE, but the functionality will not suffer because of that.

3. DAC2CORE

3.1. Contract

DAC module must:

- Make temperature, humidity and pressure readings available to the CORE;
- Accept and execute commands for hardware abstraction layer, and report back the results.
- Supply arrival and departure notifications to the CORE;
- Provide capabilities discovery.

This clearly involves two functions: broadcast, and control.

> **Note:**
> In hindsight, data collection and hardware control should have been separated into different modules. Well, in fact it is so: the broadcaster and the controller are indeed separate modules, and the only reason they exist in the same bigger module is because this is a limitation imposed by a chosen hardware platform - the 1-Wire® adapter driver is not process-safe, therefore must be accessed from a single execution context. However, it should be noted that if a different hardware used to control the HVAC equipment, the controller module becomes redundant, and everything falls into place.

3.2. Broadcast

The communication is unidirectional - from the DAC to the client. The atomic broadcast is a line, and the format of the line is trivial. Abbreviated BNF follows:

```
<dac2core-message>             ::= <capabilities-discovery>
                                 | <data-sample>
                                 | <arrival-notification>
                                 | <departure-notification>
                                 | <error-message>
<capabilities-discovery>       ::= IHAVE <device-count>: [
<device-signature> *]
<device-signature>             ::= <temperature-sensor-signature>
                                 | <pressure-sensor-signature>
                                 | <humidity-sensor-signature>
<temperature-sensor-signature> ::= T<device-address>
<pressure-sensor-signature>    ::= P<device-address>
<humidity-sensor-signature>    ::= H<device-address>
<data-sample>                  ::= <temperature-reading>
                                 | <pressure-reading>
                                 | <humidity-reading>
<temperature-reading>          ::= <temperature-sensor-signature>
<temperature-centigree>
<pressure-reading>             ::= <pressure-sensor-signature>
<pressure-mbar>
<humidity-reading>             ::= <humidity-sensor-signature>
<humidity-percent>
<arrival-notification>         ::= + <device-signature>
<departure-notification>       ::= - <device-signature>
<error-message>                ::= <device-error>
                                 | <general-error>
<device-error>                 ::= E <device-signature> <error-string>
<general-error>                ::= E <error-string>
```

Note:

For lack of better representation, **BOLD** typeface represents literals, and *emphasised* represents variable values.

For better resource utilization (and simpler code), this should have been done using multicast UDP or TCP, not unicast TCP, but the requirement to have the output debuggable (in other words, human readable) won, and as of today, the broadcaster just allows the client to open a TCP connection into itself (default TCP port is 5000), and streams the temperature readings onto the client until the client closes the connection. An added benefit of this is increased connection relibility - though even TCP connections may freeze under certain circumstances.

Note:

For convenience sake, the client is allowed to issue a "disconnect" command, which is any word starting with 'q'. Other commands will be ignored.

3.3. Control

This protocol is similar to the broadcast protocol. The communication is synchronous request-response. As the client connects to the DAC (default TCP port is 5002), the client is given a header like this:

```
IHAVE <count>: [ <address> *]
```

The <count> is a number of switch devices found by DAC, and the list of addresses follows behind the colon. For 1-Wire® devices, the address is the actual device address, followed by a colon, followed by a channel number, zero based.

There are two commands: READ and WRITE. Syntax is as follows:

```
<address> read
<address> write <value>
```

The response is either the value (for the READ command), or literal `OK` if the write was successful, or literal `E` followed by single line error message if there was either a command parsing problem or a hardware problem.

As with the broadcaster, the client is allowed to issue the "disconnect" command.

There is one issue that is not yet resolved, and is not realistic to resolve at all. Common sense tells that it is possible for the control connection to be stuck, and it is dangerous for the hardware being controlled. Therefore, a decision was made to disconnect the current control session in case when another control session request arrives. The assumption is made that the current control session is no longer accessible to initiator - it happens with TCP connections over unreliable networks sometimes.

Suppose a user starts two copies of the CORE at the same time. This will result in a race condition, when each of them trying to gain control over DAC, with the other dropping out at the same time. This situation will be aggravated if there is an intent to create a "denial of service" situation - all it takes to disrupt the normal operation is to just keep making connection attempts to the control module port.

This situation may be alleviated if the current connection is dropped only after the next connection is additionally authenticated. A simple shared secret authentication may be sufficient for this case.

Since the situation above will arise only in case of a serious system misconfiguration (such as, allowing outside access to mission critical network ports), it can probably be ignored at this time. Just keep this scenario in mind.

FIXME (VT):
Remember if this protocol supports arrival/departure notification - I think it does

4. CORE2VIEW

4.1. Contract

CORE module must:

- Provide capabilities discovery (send a complete description of a system to the VIEW);
- Notify VIEW of changes in system status;
- Accept commands from VIEW and change the system status accordingly.

4.2. Implementation

The protocol is not complicated, but the full definition will take too much time and too much place. Suffice to say that the protocol is bidirectional, with the status messages going from CORE to VIEW, and commands going from VIEW to CORE. The command is not explicitly acknowledged, nor the result is explicitly reported. However, the feedback does exist, since the command will influence the CORE status, thus triggering the notification broadcast.

A typical status notification will look like this:

```
status:unit:<unit-name>:zone:<zone-name>:\
controller:status:28.0:2.5625/2.5725/2.5625/0.01/0.0
```

This is a notification about the fact that the controller for a specified unit/zone has the setpoint of 28°C, with the error, signal, P, I, D values following.

A typical command will look like this:

```
unit:<unit-name>:zone:<zone-name>:controller:setpoint:28.0
```

This is a command to set the thermostat for the specified unit/zone to 28°C.

It is clear that a simple parser based on `StringTokenizer` is sufficient to support this protocol. The implementation is complete.

Note:
There's a caveat: it is necessary to track the GUI elements status change, followed by the command issued, followed by the status notification broadcast from the CORE. This is especially clear when the GUI elements change the status by themselves and only then issue the listener notification. One particular example is a `JSpinner`, which changes the display right away. If

> the user clicks on the `JSpinner` button twice before the command is sent to the CORE and response comes back, the oscillation loop is created, and both the core and the view enter it, which manifests itself in endless setpoint change between two adjacent values. In order to avoid this kind of behavior, some GUI elements (like `JSpinner`) must be double buffered.

5. Additional Protocol Support

As the project matures, a need for improved interoperability becomes obvious. Support for several protocols is already available:

- xAP and xPL - two protocols that used to be one, then forked.

Several protocols are considered to be implemented, but for each of them there are benefits and misses:

- BacNet - industrial strength protocol, but the specifications seem to be closed, and it may be an overkill for the puprose of this project at this point. Other consideration is that the security is not a part of this protocol specification.
- ModBus - seems to lack capabilities discovery, basically, this protocol is peer-to-peer with strictly configured peers - exactly as DZ was at earlier stages of development.
- UPnP (Universal Plug-and-Play) - this protocol is insecure by design and is full of vulnerabilities and denial of service attack loopholes. I guess this will wait until it gets fixed - which undoubtedly **will** happen, for the users have limited patience for virus attacks.
- LonTalk - I know too little about this protocol to comment at this point.
- Rendezvous - autoconfiguration protocol (a.k.a. ZeroConf), support for it is currently underway.