Reliable communication and control for the Smart Grid

Utkarsh Upadhyay

Chemin de Triaudes 11/1011,
CH-1024 - Ecublens

20 / 01 / 2012

Professeur responsable:	Jean-Yves Le Boudec
Assistant:	Dan Cristian Tomozei
Laboratoire:	LCA2

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Reliable communication and control for the Smart Grid

Utkarsh Upadhyay

Abstract—**Renewables play an increasingly important role in current energy production. Germany has set the ambitious objective of satisfying its electricity needs from renewable sources at 35% by 2020 and at 80% by 2050. The volatility of such sources renders more difficult the task of an energy supplier to match energy demand. On the other hand, the penetration of advanced meters is up from 8.7 percent in 2009 to 13.4 percent in USA. The approach to equip homes with devices which receive signals from the Distribution Systems Operator (DSO) and adapt energy consumption accordingly is showing promise. In this project we design high level communication protocols between the DSO and home device controller with a service curve based approach to demand-response. In particular, we deal with implementation issues which arise in designing and implementing such a Smart Home Controller (SHC). Various design choices made in the process of implementing the controller are discussed.**

Index Terms—**Demand Response, Service Curves**

I. Introduction

Advanced metering penetration is gaining pace rapidly in developed countries [1] and with it the need for a robust and reliable demand response algorithms and controllers is becoming apparent. There are two key demand response algorithms which have been suggested: dynamic pricing [2] and delayed supply [3]. In the former case, unregulated free markets can expose customers to high price volatility [4] which is undesirable.

For the latter case, a method of specifying demand-response constraints has been suggested in [5]. It borrows the concept of service curves [6] from Network Calculus and provides an intuitive and robust demand response mechanism. Service curves are constraints which give deterministic guarantees on the minimum amount of energy the DSO must supply in any given time interval. To make the method tractable, a special class of service curves are considered: those whose derivative is periodic. For sake of simplicity, at the risk of being misinterpreted, the period of the derivative will also be called the period of the service curve. Using their service curve contracts and the past power control signals that they received, the consumers can compute optimal load schedules using only local information. Also, DSOs are able to design and implement scheduling algorithms which can reduce the variance in the demand or reduce the peak demand while satisfying the service curve constraint for every user.

Currently, there are multiple protocols which work at various layers of the OSI stack [7]. They started with Advanced Meter Reading (AMR) protocols (e.g., IEC 61107) which were half-duplex and have evolved into Advanced Metering Infrastructure (AMI) protocols (ANSI C12.22 [8]) which can communicate in both directions. We design the high level passing of messages assuming only that the underlying layers can provide a reliable point to point communication medium on which we can build a messaging service. We concentrate on designing and implementing the SHC. A wrapper for running the applications is designed which can keep the SHC available even in face of software failures. Additionally, we give an algorithm to efficiently calculate the minimum control signal which will satisfy the service curve constraint (for a select class of service curves) and prove its correctness. We also make some design recommendations for the DSO. Under these design assumptions, we show that the algorithm is also efficient.

In Section II we give briefly describe our solution: architecture of the controller, the prediction algorithm, the wrapper and the implementation. Section III discusses some alternate design of the architecture and explains the choices made. In Section IV, we describe major contributions made in this work. Section V is about the value of the project as an educational endeavor and Section VI presents a self-assessment of the milestones reached. Section VII describes the current state of the implementation and the road ahead.

II. Smart Home Controller

In this project the aim is to design reliable communication protocols between the DSO and SHC and to implement a reliable SHC which can listen to control signals from the DSO and adapt the consumption of the house accordingly. The SHC should also be able to verify that the service curve constraints are being met. In the following sections, we describe the primary components of the solution. First, we give an overall architecture of the SHC and various communication channels. Then we briefly describe our wrapper which can bring some reliability to software executed inside it. As a third part, we present our solution to the problem of predicting minimal future signals and finally the implementation.

A. Software and Communication Architecture

The design of our SHC is highly modular to account for changes in the requirements and because it allows the developers to use the best tools for the tasks without worrying about interoperability. For instance, in the present implementation, we have used C++ for communicating with the Smart-plugs as the proprietary library was provided in C++, Ocaml for the core controller because of type-safety and availability of native-code compilers, Python for developing the UI, and ZeroMQ[1] for communicating between the modules. Also, the modules can be run on different machines (or cores) for dealing with scalability problems.

Figure 1 gives a broad overview of the architecture of the communication protocol and the parts implemented. The arrows indicate communication between different modules. The development was done for two actors: DSO and the home user. The home user is able to interact with the SHC using a UI while the DSO communicates with it directly. Though recording history is important for the user, it can be done separately. This makes the main controller simpler and, more importantly, avoids issues such as running out of memory which can effect the stability of the controller. The controller, nevertheless, does need some memory to save its state so it may recover from unexpected restarts. In Section III-B, we give our recommendations which will help us in ensuring that this does not happen.

The *Metering* module is kept separated from the Controller as a homogenising layer between the Controller and the devices. Devices generally come coupled with custom controllers: they can have direct wired controls (some HVACS), they could be SE profile or HA profile compliant ZigBee devices [9], or any other method. For example, for the purpose of testing our implementation, we used ZPlugs from Cleode which are SE profile compliant smart-plugs which can monitor usage and turn on/off devices connected to them. These devices come with their own firmware and libraries which are usually proprietary and may be unstable. Instead of letting the Controller handle these details, the Metering module takes responsibility of keeping these devices up and running while providing SHC with regular updates on their performance. This module will take care of failures of the library/devices and take appropriate actions.

However, either due to software bugs or because of misbehaving underlying libraries, this layer can suffer from Byzantine failures [10]. When such problems are detected, it should be possible to terminate and restart the application, with the hope that it would resume correct behaviour upon restarting. Because of this separation between the SHC and the Metering module(s), the SHC layer can undertake such a step after the failure is detected either by the user or by the Controller.

B. Reliability

The SHC and the Metring module(s) are executed inside a wrapper which can deal with two common problems:

- Software crashes (including in proprietary libraries)
- Unresponsive systems

Further, the wrapper can be asked by a third party to kill and restart the underlying application if the application is showing Byzantine behaviour.[2]

C. Prediction

Predicting the control signal which will satisfy the service curve constraint is an important problem for both the users as well as the DSO. However, the naive algorithm of computing a max-plus convolution for each time point in the future is impractical for a system which operates in continuous real time. Though this problem can be partially addressed by making the time-steps discrete, we chose to keep time continuous. This design choice is discussed in Section III-B.

In this work, we formulate a representation scheme for the inputs/outputs of the prediction procedure (i.e. the control signal, the service curve and the prediction itself) which is easy to manipulated and communicate. Based on this representation, we give an algorithm to produce these predictions. We also analyse the complexity of our algorithm and recommend enforcing reasonable conditions under which the algorithm is efficient (in Section III-B).

D. Implementation

We implemented a prototype of our SHC trying to keep it as deployment ready as possible. This actual implementation helped us tune our design and made us look more closely at our algorithms and issues involved in implementing them. The implementation is fully functional and can be used as a test-bed for experiments.

III. Design choices

Design of any software is more an art than science. During development, we made many design choices and it is difficult to determine whether they were the best choices at this stage. In this section, I attempt to argue their pros and cons and present alternate design patterns which could have been employed. These sections also contain some recommendations for future work.

[1]www.zeromq.org

[2]Shoot The Other Node In The Head (STONITH) principle

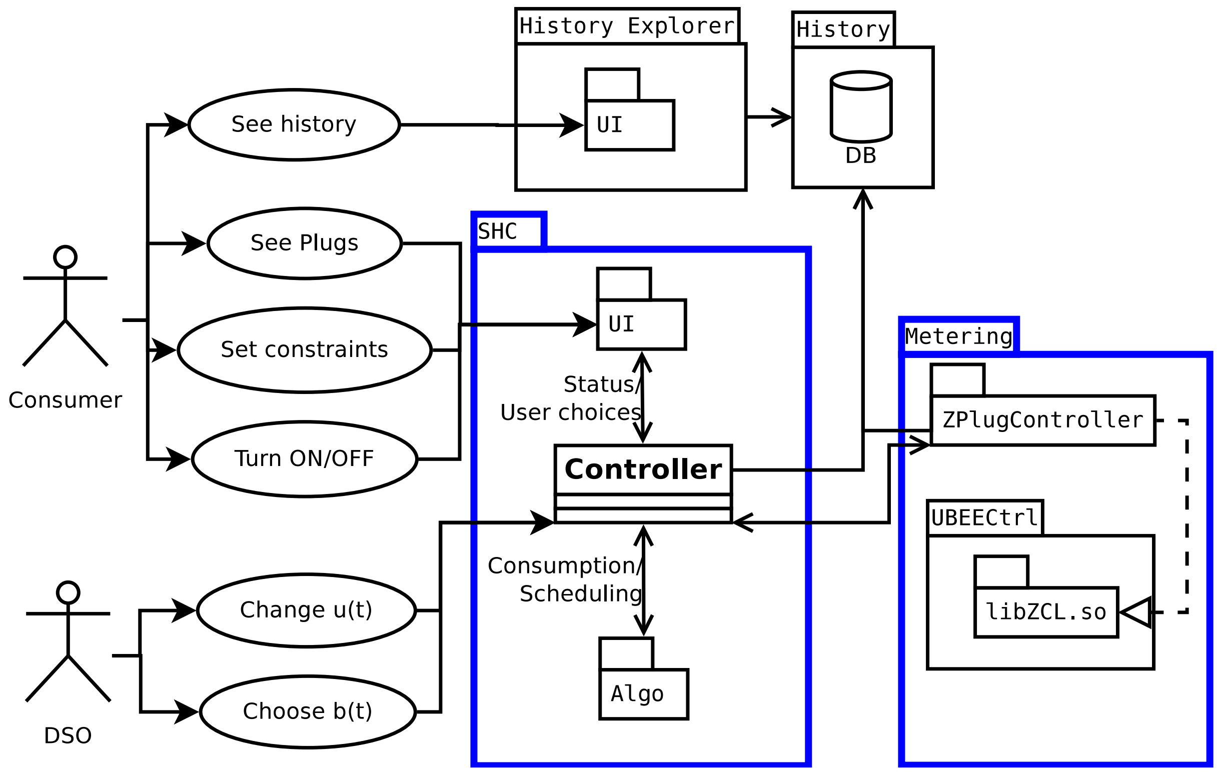


Fig. 1: Design of the controller. Bold lines show the parts implemented. $b(t)$ is the service curve and $u(t)$ is the control signal. **DB** stands for Database used to store the entire history. `libZCL.so` is the proprietary library for the Smartplugs we use and `UBEE` and `ZPlug` are their trademarks. There could be more than one *Metering* and *UI* modules for a single controller

A. *PUB-SUB v/s REQ-REP*

In our design, updates have to be sent by the server to clients in three instances:

- DSO to SHCs
- SHC to Algorithm module and UI(s)
- Metering module to SHC

Choosing which protocol to use in these conditions is important as it effects:

1) Amount of data transferred over links
2) Clients' recovery after a failure
3) Complexity of the protocol

In the REQ-REP design, the clients would *pull* the needed updates from server while in the PUB-SUB design, the server *pushes* updates to the clients. We chose the PUB-SUB design for all three cases. Both approaches have many pros and cons and we give only the criteria which we deemed most important:

1) **Overloading**: The REQ-REP model can very easily overload the servers as the number of clients increases. The PUB-SUB model is easier to scale in these settings because message relays can be put in place in the topology without changing the server (See Figure 2). The converse problem in the PUB-SUB model is that the publisher might overload the subscribers. In the DSO pushing updates to SHC case, stability of the DSO is arguably more important than that of the clients. Similarly, in the case of SHC pushing data to UIs, the stability of the server is more important. With Algorithm module and the Metering module, the questions of whose stability is more important are moot.
2) **Separation of concerns**: While scaling the DSO to a large customer base, one of the concerns would be to scale the communication which happens between them. We would need to modify the DSO's application to address this issue in the REQ-REP model, while in the PUB-SUB model, we can scale the communication in messaging layer separately from the application. Similar arguments can be

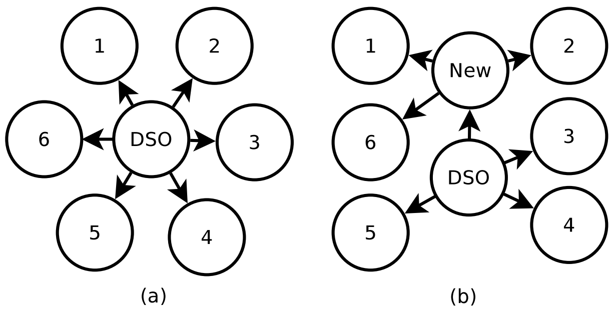


Fig. 2: Solving the scalability problem with one extra node (Load on the DSO reduces from 6 nodes to 4 nodes)

made in the case of SHC to UIs communication, as there can be multiple interfaces for a single SHC.

3) **Logging and diagnostics**: These should be as non-intrusive as possible and can require different quality of service. In the REQ-REP model, the server has to deal with them individually at the application layer but in the PUB-SUB model, their QoS can be controlled at the messaging layer.
4) **Protocol Complexity**: The PUB-SUB model is easier to reason about than the REQ-REP model in the event of failures in clients and message drops.

Now we discuss the cases in more detail.

1) DSO → SHC: A drawback of the PUB-SUB model is that the SHCs which have just started (either for the first time, or after a failure) will have no means of obtaining the current state (the service curve chosen and the current control signal) other than to wait for the next update from the server. Hence, the DSO should periodically push updates containing entire state information. This problem can be avoided in the REQ-REP model as new clients can request the state from the server when they become active.

Nevertheless, because of the distributed nature of the demand response algorithm, very little information needs to flow from the DSO to the SHC. In future deployments, if the size of the state becomes large, we may need a more hybrid protocol and separate communication channels for the control signal and the service curve.

2) SHC ↔ Metering Module: In this case, the Metering module sends regular updates to the SHC about the various devices' consumption. Here, the advantage in having a PUB-SUB model is that the Metering module(s) can decide their frequency of updates instead of the frequency being hard-coded in the SHC.

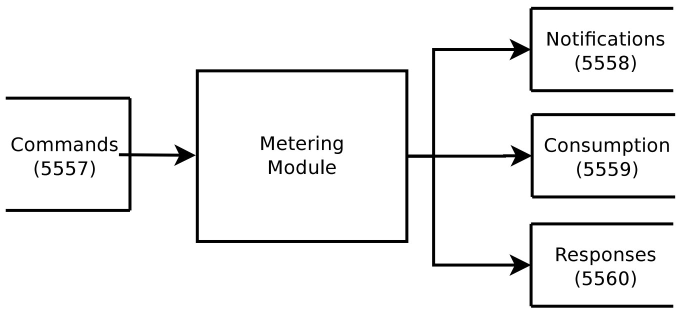


Fig. 3: Metering module (with example ports)

However, in this case, the Metering module can be instructed by the Controller to take action on devices. The communication scheme is shown in Figure 3. Currently, the actions include only turning on/off devices. In such settings, it is ensured that the each command sent to the module will have one reply. This communication scheme can be implemented via a REQ-REP model. However, for allowing diagnostic probes, we have implemented it as a REQ-PUB scheme, i.e. SHC sends command to the Metering module and the Metering module publishes its response. An advantage of this scheme is any spurious command senders get to know of presence of other senders when they see response to commands which do not originate from them. This can be used as a cheap uniqueness detector.

In future, the updates will contain more information about the devices. For example, the elasticity in its demand (if it is a battery) or for how long it can be delayed (if it is a thermostat). Such information can be included in the messages and handled in the SHC easily.

B. Discrete v/s continuous time

In previous work [5], time was discretized on the consumer's side (at the SHC) to make the problem mathematically tractable and for ease of presentation. One of the algorithms for DSO side allocation policy [5, Alg. 1] explicitly relied on time being discrete. However, during implementation, we realised that the sampling period becomes an important parameter not only for determining accuracy, but also for determining the scalability of the system. For example, if the period of the service curve is large and the sampling time is small, prediction can become computationally expensive. On the other hand, if the sampling period is large, the predictions can become too inaccurate. An advantage of this scheme, however, is that the computational time, though invariably high, is independent of input.

However, we took the alternate route and developed an algorithm for accurate prediction in continuous time. Our representations of the curves was amenable to it and it was intellectually satisfying to eliminate one parameter from the system without sacrificing anything in return. We also proved the correctness of the algorithm. However, the algorithm's performance is now dependent on the input signal. Hence, we have two recommendations to keep the algorithm efficient (see Section IV-C for details):

1) There should be an upper limit on the number of times the DSO can change its signal in one period of the service curve.
2) There should be an upper bound on the number of different segments the service curve can contain.

With these two limitations, efficiency of the protocol can be guaranteed. Limiting the size of service queues (condition 2) also has the desirable consequence that it limits the amount of data that SHC would need to maintain for recovery.

C. Choice of third-party tools

Ideally, the SHC should not depend on any third party tools to function. However, to keep the implementation simple, we used some lightweight tools which can be embedded with the SHC when needed.

1) Messaging service: Instead of developing our own communication from scratch, we chose to work with ZeroMQ [11, 12], a messaging service which works atop of layer 4 protocols. This eases communication between modules and lets developers focus more on the application. For instance, the PUB-SUB model is provided as a basic service with ZeroMQ. We chose ZeroMQ over other messaging services because it is broker-less and decentralized. This reduces possible points of failures and makes local deployments easier and is an ideal framework to mimic for the AMI protocols.

2) Persistence layer: We used SQLite[3] to save the state of the system and recover from unexpected restarts. Other options we considered were writing to disk manually or having a separate local server to save the complete history as well as the last state. Maintaining a separate server is not a good design because it would require additional guarantees of availability of the server when the controller restarts, which might actually be more difficult than restarting the SHC itself. Also, writing manually to disk using primitive operations is a fragile operation which might leave the disk in a corrupted state if interrupted at the wrong time (e.g., in case of a power failure). The SQLite library is designed to be embeddable, is very small in size (300 KB), and is reputed to be extremely resilient to failure even in extreme cases such as power failures. We utilize the atomic transactions to ensure that we always leave the system in a consistent state.

IV. Contributions

Before explaining the contributions, we will define some terms and lay some groundwork.

Definition 1. *Function $b(\cdot) : \mathbb{R}^+ \to \mathbb{R}^+$ is a **service curve** iff:*

1) $b(t) = 0$
2) $\forall\, t, s \geq 0.\, b(t) + b(s) \leq b(t+s)$

Property 2 is also called the super additive property.

This is the general definition of service curves. However, we consider only a subclass of these curves, which we define below.

Theorem 2. *If $\gamma(t) : [0, t'] \to \mathbb{R}^+$ is a convex function such that $\gamma(0) = 0$ and*

$$\beta(t) = \lfloor \frac{t}{t'} \rfloor \gamma(t') + \gamma(t - \lfloor \frac{t}{t'} \rfloor \cdot t')$$

then $\beta(t)$ is a service curve.

Proof: Proof omitted for brevity. See appendix. ■

[3]www.sqlite.org

Definition 2. *A service curve $\beta(\cdot)$ defined as in Theorem 2, will be called a **special service curve**. The value t' will be called the **period** of the special service curve.*

Also useful for us would be the following definition of the energy received since the service curve constraints actually are on increments of this function:

Definition 3. *The **total energy function** $U(t)$ is the integral of the control signal $u(t)$ sent by the DSO.*

$$U(t) = \int_0^t u(\tau)\, d\tau$$

With these definitions, we are in a position to understand the theoretical contributions made in the present work.

A. Theoretical Contributions

While the previous results laid sufficient groundwork for designing an implementation of the demand response system based on some service curves, we were able to generalize the results for a richer set of service curves. Previous work [5] considered:

1) *Special service curves* derived from:

$$\gamma(t) = \begin{cases} z_{min} t & \text{if } t < t_0 \\ z_{max}(t - t_0) + z_{min} t_0 & \text{if } t_0 \leq t \leq t_0 \end{cases}$$

with period t_1, $0 \leq z_{min} \leq z_{max}$. These were referred to as *binary service curves.*

2) Control signal which is bounded above by z_{max}

Under these restrictions, the following theorem was proved:

Theorem. *Let $u(t)$ be a sequence of control signals defined up to some time horizon T. Assume that $u(t) \leq z_{max}$ for all $t \leq T$. The two properties are equivalent:*

1) $\int_t^{t'} u(\tau)\, d\tau \geq \beta(t - t')$ *for all $t' < t \leq T$,*
2) $u(\tau) \geq z_{min}$ *for all $\tau \leq T$, and* $\int_t^{t+t_1} u(\tau)\, d\tau \geq z_{max} t_1 - (z_{max} - z_{min}) t_0$ $\forall\, t$ *such that* $t + t_1 \leq T$.

where $\beta(\cdot)$ is a special service curve constrained by condition 1.

Proof: See [5, Thm. 2]. ■

This result is useful in providing a practical method for enforcing and verifying the service curve constraint β. It is sufficient to check that the control signal satisfy $u(\tau) \geq z_{min}$ at every time slot τ and to keep in memory the history of the control signals over the last t_1 time units; their integral should never drop below $z_{max} t_1 - (z_{max} - z_{min}) t_0$.

In this work, we extend this result to a *special service curve* derived from any star shaped $\gamma(\cdot)$ and remove the upper bound restriction on the control signal by proving the following simpler result:

Theorem 3. *If control signal $u(\cdot)$ violates special service curve $\beta(\cdot)$ for the first time at time T, then*

$$\exists\, t \in [T - t', T).\, U(t) + \beta(T - t) > U(T) \qquad (1)$$

Proof: Omitted here for brevity. See appendix. ■

Equation (1) essentially states that if a service curve violation happens at all, then there is a point in the interval $[T - t', T]$ for which the service curve condition is violated. In other words, if there is no violation of the service curve in the time $[T - t', T]$ then there are no violations. Using Theorem 3, we can check for the *first* violation of the service curve using only the history over one period t' of the control signal.

It should be noted here that verifying whether control signal satisfies a general service curve requires computing a max-plus convolution [6, p. 155] while in the simpler scenario considered in [5], verification required calculating an integral. Calculating an integral is a simpler operation than calculating a max-plus convolution. Doing the more complex of the two operations is a sacrifice to allow for more general service curves. However, under the recommendations made in Section III-B and using our method of representing the *special service curve* and the *total energy function* described in the next section, calculating the max-plus convolution is also computationally efficient.

B. Data formatting

This section describes some of the issues in implementing the SHC and it precedes the prediction algorithm section because the algorithm depends on how various values are represented in the SHC.

1) Communication of values: We assume that instead of exercising continuous control (by continuously manipulating the control signal $u(t)$), the DSO exercises its control by sending discrete messages to the SHCs. This is a reasonable assumption to reduce the complexity of the communication protocol and to ensure scalability of the DSO. For the sake of simplicity, in the implementation the messages are sent in plain-text JSON format.

The content of the message is[4]:

Command:	DSO
`CHANGE_TO`	Next control signal

Similarly, the DSO can issue commands with the following content to alter the service curve:

Command:	SERVICE_CURVE
[(slope, len.)]	List of slope, length of segments.

[4]The actual message has an extra field, and is described fully in the appendix

These messages dictate how the related curves are represented in the SHC. Details of other message formats can be found in the appendix.

2) Representation of curves: Here, we describe how the curves are represented in the Controller after translating them from the DSO's messages.

Control signal: As we assume that the DSO informs of any changes in the control signal by discrete signals, the *total energy function* $U(t) = \int_0^t u(\tau)\, d\tau$ will have constant slope between two changes. Hence, we can represent the *total energy function* from the DSO as a piecewise-linear function. The slope of the function at time t is the power promised by the DSO at time t. Internally, to represent such a curve we store the times t at which the control signal changes and the corresponding value of $U(t)$. For complete representation, we also store the value at the present time T. We will use the symbol $\mathbf{T}_f$ to denote the sorted list of times at which the slope of the piecewise linear function $f(\cdot)$ changes. If there are K points in the list $\mathbf{T}_U$, then $\max(\mathbf{T}_U) = t_K = T$. In the algorithm (and in proofs which relate to the algorithm), the present time is denoted by t_K to match programmatic notation. The notation $U(.)$ would be used for the list of tuples $(t_i, U(t_i))$. For example, the *total energy function* shown in Figure 4(a) is represented as $U(\cdot) = [(11, 10), (11.5, 11), (13, 12))]$.

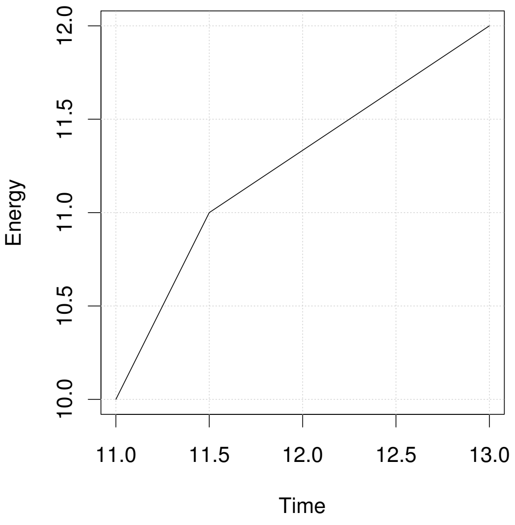
A corollary of Theorem 3 is that to predict the minimum signal at time $t > T$, it is sufficient to know $U(t)$ from $[t - t', T]$. Hence, we can truncate the control signal from $T - t'$ to preserve memory. To calculate $U(t)$ retrospectively, we find $t_{i-1}, t_i \in U$ such that $t_{i-1} \le t < t_i$ and linearly interpolate between $U(t_{i-1})$ and $U(t_i)$.

Service Curve: Though the service curve can be any function which satisfies the required constraints given in Theorem 2, in practice the service curve is usually a piecewise linear function. Otherwise, we can approximate the actual service curve by a piecewise linear function (but restrictions given in Section III-B may limit the accuracy). Such a representation of the service curve is easy to communicate, store and manipulate. The representation of the function $\beta(\cdot)$ will be similar to that of $U(\cdot)$ described above. We need to save only one period of the service curve for the algorithm. For example, the *special service curve* shown in Figure 4(b) is saved as $\beta(\cdot) = [(0, 0), (1, 0), (3, 2)]$.

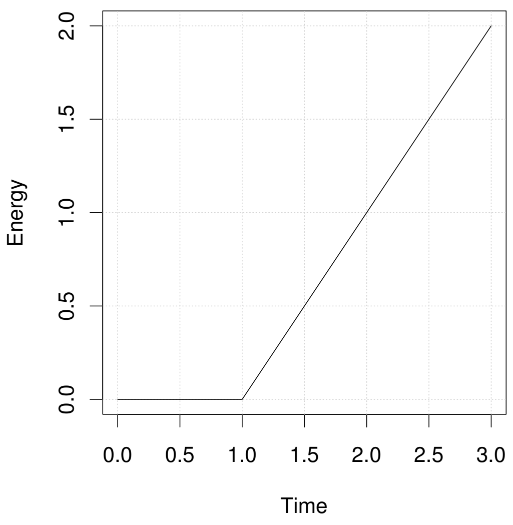
C. Prediction Algorithm

Before describing the theorem, we define the term *cover* for the purpose of our algorithm as follows:

Definition 4. *For a given segment on the total energy function $(\tau_1, U(\tau_1)) - (\tau_2, U(\tau_2))$ and a* special service curve *$\beta(\cdot)$ with period t', a* **cover** *$C(\cdot)$ is defined for the*



(a) Sample Total Energy function



(b) Sample *special service curve* with period 3

Fig. 4: Examples of curves

time period $[\tau_1, \tau_2 + t']$ *as:*

$$C(t) = \begin{cases} U(\tau_1) + \beta(t - \tau_1) & \text{if } \tau_1 \le t < \tau_1 + t' \\ U(t - \tau_2) + \beta(t') & \text{if } \tau_1 + t' \le t < \tau_2 + t' \end{cases}$$

The algorithm shown in Procedure 1 is used to predict what is the minimum energy which must be supplied by the DSO to comply with the service curve contract. For simplicity, it is assumed that the *total energy function* is known for at least one period of the *special service curve*. Extending the algorithm to cover the other case is trivial.

Procedure 1 Predicting guaranteed signal

```
Input: $\beta(\cdot) = [(s_1, \beta(s_1)), \ldots, (s_N, \beta(s_N))]$,
    $U(\cdot) = [(t_1, U(t_1)), \ldots, (t_K, U(t_K))]$
Output: predicted segments/function
 1: $\hat{V}_1$ = $[(t_K + s_i, U(t_K) + \beta(s_i))$ for $s_i$ in $\mathbf{T}_\beta]$
 2: scTip = $(t_K,\ \beta(s_N) + U(t_K))$
 3: for $j$ = $K - 1$ to $1$ do
 4:     $t_j$ = $\mathbf{T}_U[j]$
 5:     C = $[(t_j + s_i, U(t) + \beta(s_i))$ for $s_i$ in $\mathbf{T}_\beta]$
 6:     C.add(scTip)
 7:     $\hat{V}_{(K-j+1)}$ = maxCover($\hat{V}_{(K-j)}$, C)
 8:     $\hat{V}_{(K-j+1)}$.limitTime($t_K$, $t_K + s_N$)
 9:     scTip = $(t_j + s_N,\ U(t_j) + \beta(s_N))$
10: end for
11: return $\hat{V}_K(\cdot)$
```

In line 1, we calculate the first approximation of the prediction ($\hat{V}(\cdot)$). We will improve this iteratively to arrive at the final prediction. In *scTip*, on line 2, we keep track of the *tip* of the service curve, as a (time, energy) tuple, if drawn from the last observed control signal. This variable would keep track of the *tip* of the service curve drawn from ends of segments of $U(\cdot)$. In the lines 3–10, we improve our approximation of the prediction. In line 5, we calculate a *new cover* by calculating the minimal values which would satisfy the service curve from the time t. Finally, we add the last segment which joins the current *tip* to the previous *tip* (line 6). This makes $C(\cdot)$ a *cover* by definition. Figure 5(a) shows an example of a first iteration of the algorithm.

We improve the prediction by taking the maximum of the new cover and the old prediction (lines 7). `maxCover` takes as input two functions $f(\cdot)$ and $g(\cdot)$ and returns a function $h(\cdot)$, such that $h(t) = \max\{f(t), g(t)\}$ if both $f(t)$ and $h(t)$ are defined, $h(t) = f(t)$ if only $f(t)$ is defined and $h(t) = g(t)$ if only $g(t)$ is defined. Line 8 limits the time domain of the resulting function to only future time values. Figure 5(b) shows the resulting predictions after the sample iteration of Figure 5(a). Finally, we calculate the *tip* of the service curve for the next cover, save it in *scTip* (line 9) and the loop continues to the next segment of $U(\cdot)$.

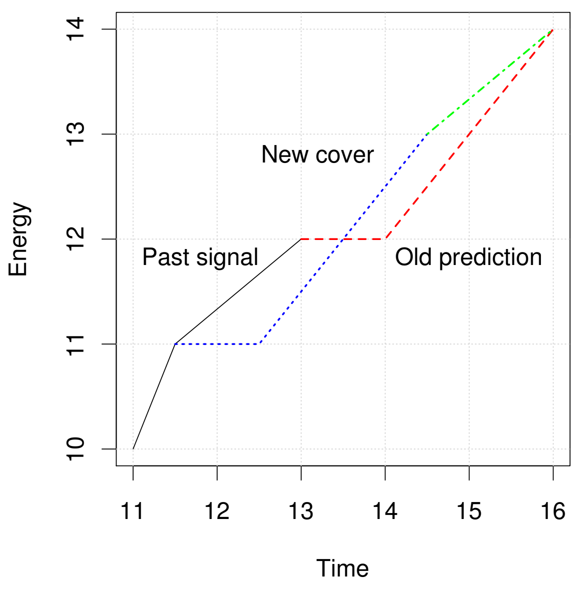
To show that this algorithm is correct, we need to first define the following function:

Definition 5. *For a given energy function* $U(\cdot)$ *defined till time* T *and a given* special service curve $\beta(\cdot)$ *with period* t'*, define* ***ideal prediction*** $\hat{U}(\cdot)$ *as:*

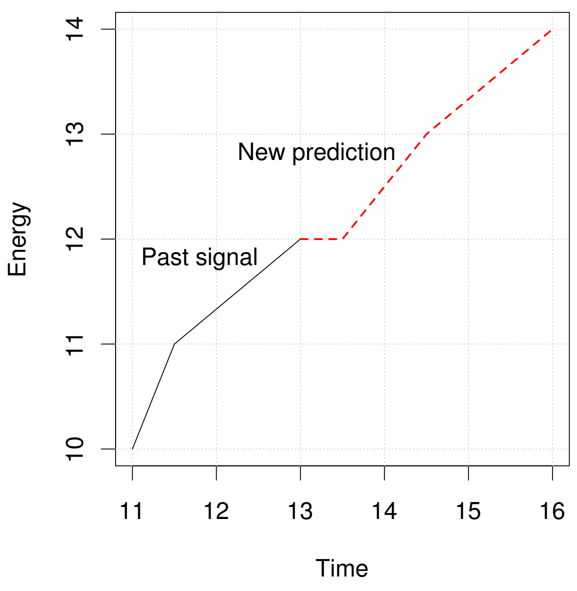
$$\hat{U}(t) = \sup_{s \in [t-t', T]} \{U(s) + \beta(t - s)\}$$

for $T < t \le T + t'$.

This function has the useful property that for every function $V(\cdot)$ which has value less than $\hat{U}(t)$ at any $t \in [T, T + t']$, there will be at least one point $s \in [T - t', T]$ such that $V(t)$ violates the service curve for s.



(a) First iteration of the algorithm before line 7



(b) Prediction after the first iteration after line 8

Fig. 5: An iteration of the algorithm

Now we state the main result of the section which proves that the output of the algorithm is correct.

Theorem 1. *For a given* total energy function $U(\cdot)$ *defined till time T and a* special service curve $\beta(\cdot)$ *with period t', the output of algorithm 1 is the minimal function which will satisfy the service curve constraint.*

Proof: To prove this result, we need some intermediate claims. We start by first proving that the *ideal prediction* function is identical to the prediction. We show this by representing both functions at each time as maximum of a finite number of quantities. All the intermediate proofs can be found in the appendix. First for the *ideal prediction*:

Theorem 4. *Given $U(\cdot)$, an energy function, and $\beta(\cdot)$, a* special service curve *with period t', then for $t \in [t_K, t_K + t']$:*

$$\hat{U}(t) = \max \begin{cases} U(t-t') + \beta(t'), \\ U(t_I) + \beta(t - t_I), \\ U(t_{I+1}) + \beta(t - t_{I+1}), \\ \vdots \\ U(t_K) + \beta(t - t_K) \end{cases}$$

where:

- *I is chosen such that $t_I \leq t - t' < t_{I+1}$, $t_i \in \mathbf{T}_U$,*
- *$t_K = \max(\mathbf{T}_U)$, and*
- *$\hat{U}(\cdot)$ is the* ideal prediction *for $U(\cdot)$ and $\beta(\cdot)$.*

Similarly, for the output of algorithm 1:

Theorem 5. *Given $U(\cdot)$, an energy function, and $\beta(\cdot)$, a* special service curve *with period t', then for $t \in [t_K, t_K + t']$:*

$$\hat{V}(t) = \max \begin{cases} U(t-t') + \beta(t'), \\ U(t_I) + \beta(t - t_I), \\ U(t_{I+1}) + \beta(t - t_{I+1}), \\ \vdots \\ U(t_K) + \beta(t - t_K) \end{cases}$$

where:

- *I is chosen such that $t_I \leq t - t' < t_{I+1}$, $t_i \in \mathbf{T}_U$,*
- *$t_K = max(\mathbf{T}_U)$, and,*
- *$\hat{V}(\cdot)$ is the value returned after executing the algorithm 1 with $U(\cdot)$ and $\beta(\cdot)$ as input.*

Finally, we show that the *ideal prediction* function satisfies the service curve constraint.

Theorem 6. *Given a total energy function $U(\cdot)$ defined for $t \leq T$ and a special service curve $\beta(\cdot)$ with period t', their* ideal prediction *function $\hat{U}(\cdot)$ defined in Definition 5 satisfies the service curve constraint.*

Using theorems 4 and 5, we conclude that the output of the algorithm and the *ideal prediction* are the same function. By theorem 6, we know that the output of the algorithm satisfies the service curve and hence, by construction, the ideal prediction function is the minimal function which will satisfy the service curve.

Hence, proved. ■

Complexity: Let N be the number of segments in the *special service curve* $\beta(\cdot)$ and K be the number of times the control signal changes in the time $[T - t', T]$. Calculating the first approximation takes $O(N)$ time on line 1. Now consider the loop between lines 3–10. It executes $O(K)$ times. Line 5, where the new *cover* is computed, takes $O(N)$ time. The function *maxCover* and *limitTime* (called on line 7 and 8) can be implemented such that the calls to them take $O(N)$ time as well (see appendix for such an implementation). Hence, the overall complexity of the algorithm is $O(N \cdot K)$. This is the basis behind the two recommendations made in Section III-B. If both N and K are bounded be design, then the algorithm is guaranteed a fixed worst case performance.

D. Implementation

The implementation uses multiple third party tools and spans three different programming languages. In total it took about 7,000 lines of code. Some details are given in Table I.

Language	Lines of code	Comments	Component
Python	4,300	1,300	UI, Test harness
C++	800	220	ZPlug wrapper
Ocaml	2,000	460	SHC
Total	7,100	2,080	–

TABLE I: Details about the code in the implementation

- **C++**: The library provided by Cleode for ZPlugs was in C++. Hence, it was a natural choice for the Metering module.
- **Ocaml**: Used because of the excellent trade-off it provides between performance and correctness. Ocaml programs are type safe and, hence, avoid many problems of dynamically typed languages. Further, there are tools available [13] which can formally prove certain properties of the system. On the performance front, well-written Ocaml code compiled with native byte-code compilers can perform as well as C code. This can be useful while porting the application to a embedded system.
- **Python**: Used for its ease. The UI and the test harness is written in Python.

As stated before, we used ZeroMQ for communicating between the modules and SQLite for saving the state of the SHC. ZeroMQ has bindings for all three languages we used (and is itself written in C++) and SQLite has bindings for Ocaml. The code is maintained on a SVN repository and its development can be traced via the regular commit logs. Internally, the design of the controller is such that it is easy to add handlers for more commands.

Also, a testing harness along with a large array of tests has been developed in tandem with the SHC and the Metering module. These tests, made on lines of unit tests, alone take about 1,500 lines of Python code and are written to test the respective module's commonly used API. Regularly running these tests ensures that the basic functionality provided by the implementation is correct and make incremental development easy.

E. Wrapper design

To ensure reliability and to meet the design requirements, we have introduced an object monitoring wrapper layers above the controller. The inspiration behind this wrapper was the design of FALCON described in [14]. This layers solves the following three problems:

1) Software crashes (the process exits with an error)
2) Live-locks or infinite loops (the process stops making progress)
3) Killing and restarting the application on request (to handle Byzantine faults)

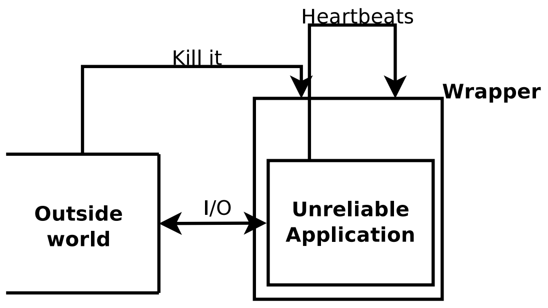


Fig. 6: Design of the wrapper

Figure 6 shows the design of the wrapper. The wrapper executes the application and expects regular heartbeats from it. If the application crashes, with a nonzero error code, the wrapper would automatically restart it. If the application stops generating heartbeats, then the wrapper first kills it using appropriate signals to the underlying OS, and then starts a new copy of the application. These heartbeats are currently communicated using `tcp` ports with ZeroMQ instead of the Unix specific inter-process communication sockets.

Finally, the wrapper can kill the unreliable application and restart it upon request. This feature essentially shifts the burden of identifying the Byzantine fault onto the *outside world* (See Figure 6). In our case, both the Metering application as well as SHC runs under such a wrapper. Hence, the user can remotely restart the SHC using the UI and some safeguards can be put in place in the SHC to restart the Metering module. This communication also, for simplicity, relies on ZeroMQ and `tcp` ports.

The wrapper is written in python and is independent of the modules themselves. It could be used for ensuring high availability of any application[5].

[5]It is hosted here https://github.com/musically-ut/pyoolproof

V. Educational experience

On a conceptual level, the project was fruitful in two aspects:

- I learned about existing Demand Response mechanisms, their shortcomings and the elegance of using service curves to define the contract. I believe that the service curves are only half the story, as there are many constraints which cannot be stated in their form and perhaps it is possible to devise more intuitive structures over it. This area has challenging open questions while having direct benefit for society at large. I find it to be a perfect avenue for research.
- With respect to implementation I learned much about good designing and how to think about scalability, fault-tolerance and other latent issues in the prototype stage itself and I was surprised to learn that it is nearly impossible to guarantee reliability. I learned about how software is tested and verified for applications like avionics and found that the process relies much more on human reviewing than on automated verification.

On the concrete level, the project required a good mix of practical as well as theoretical skills. I practised my software engineering skills in making sure that the project is easy to develop and maintain for future developers and I used my analytical skills to formulate and prove correctness of the prediction algorithm. I had dabbled about in functional programming before but while executing this project, I was able to immerse myself in a full scale study of Ocaml. I learned many details of how functional and object oriented aspects of the language interact. Further, the project helped me gain some familiarity with network calculus.

VI. Self Assessment

I am satisfied with the implementation in place. I like the overall design, its flexibility and the choice of tools for the task. I would have like to use more fuzzy testing on the wrapper using Library Fault Injector [15]. Though I have written unit test for testing for common cases, they could have been more rigorous: the code coverage attained by the unit tests is only about 50% for my unit tests (an estimate). For comparison, in most avionics software, it is necessary to have 100% code coverage.

The implementation task went along smoothly which left me ample time to work on verifying the implementation towards the end. However, it was only in early December that we realised that the coming up with a prediction algorithm would be a non-trivial exercise. Though I had been stress testing the SHC and Metering module nearly every night since their first working version, I intentionally delayed the systematic rigorous tests until after I had found a correct and efficient prediction algorithm. I found the generalization of the results and the algorithm to predict the future control signals enjoyable but they turned out to be slightly more time intensive then I had planned for. I would have liked to spend some of that time in designing more systematic stress tests for the system.

Barring the rigorous testing phase, I believe I was able to fulfil the requirements of the project fully.

A. *Future work*

It would be apt to include in which directions the work can continue. One aspect of the design which clearly deserves some attention is security. The importance of security cannot be overstated since the costs of impersonation and leaking of information of consumption can be immense.

One of the reasons for choosing Ocaml was possibility of formally verifying correctness of parts of the SHC. Also, stress testing of the system with LFI is a possible next step.

On the theoretical side, one could attempt to improve the algorithm to be incremental. This would be particularly useful for the DSO, which may have to keep estimates for all its consumers for optimization purposes. The optimization methods themselves offer a vast area of research work.

VII. Conclusion

In this project we have implemented a Smart Home Controller which receives signals from the DSO and is able to:

- Detect violations of the *special service curve* constraints.
- Predict the minimal future signals and display them for the user.
- Reduce consumption at the user's end if the demand exceeds current capacity.
- Recover from common component failures or software errors.

Also, we were able to generalize the results of previous work and give a new algorithm for prediction of worst case signal which is efficient and proven correct. This algorithm is implemented and used in the SHC in displaying the predictions to the user. This application is ready to be deployed and used as a testbed for future experiments.

References

[1] Federal Energy Regulatory Commission, "Assessment of demand response & advanced metering." http://www.ferc.gov/legal/staff-reports/11-07-11-demand-response.pdf, November 2011.

[2] S. Borenstein, M. Jaske, and A. Ros, “Dynamic Pricing, Advanced Metering, and Demand Response in Electricity Markets,” Mar. 2002.

[3] OECD, “Oecd economic surveys: France 2011..” OECDiLibrary, March 2011.

[4] S. Meyn, M. Negrete-Pincetic, G. Wang, A. Kowli, and E. Shafieepoorfard, “The value of volatile resources in electricity markets.” Proc. of the *49th Conf. on Dec. and Control*, 2010.

[5] J.-Y. Le Boudec and D.-C. Tomozei, “Demand Response Using Service Curves,” in *The second European conference on Innovative Smart Grid Technologies (IEEE ISGT-EUROPE 2011)*, 2011.

[6] J.-Y. Le Boudec and P. Thiran, *Network calculus: a theory of deterministic queuing systems for the internet*. Berlin, Heidelberg: Springer-Verlag, 2001.

[7] K. De Craemer and G. Deconinck, “Analysis of state-of-the-art smart metering communication standards,” March 2010.

[8] A. Moise and J. Brodkin, “Rfc:6142 “ansi c12.22, ieee 1703, and mc12.22 transport over ip”.” http://tools.ietf.org/html/rfc6142, March 2011.

[9] Z. Alliance, “Zigbee standards overview.” http://www.zigbee.org/Standards/.

[10] L. Lamport, R. Shostak, and M. Pease, “The byzantine generals problem,” *ACM Trans. Program. Lang. Syst.*, vol. 4, pp. 382–401, July 1982.

[11] P. Hintjens, “ZeroMQ: The Guide.” http://zguide.zeromq.org/, 2010.

[12] M. Sûstrik, “ØMQ: the theoretical foundation.” http://www.250bpm.com/concepts, July 2011.

[13] P. M. Rondon, M. Kawaguci, and R. Jhala, “Liquid types,” *SIGPLAN Not.*, vol. 43, pp. 159–169, June 2008.

[14] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish, “Detecting failures in distributed systems with the FALCON spy network,” in *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2011.

[15] P. Marinescu and G. Candea, “Lfi: A practical and general library-level fault injector,” in *Dependable Systems Networks, 2009. DSN ’09. IEEE/IFIP International Conference on*, pp. 379 –388, 29 2009-july 2 2009.

Appendix

The proofs for the theorems given above are presented here.

Theorem 2. *If $\gamma(t) : [0, t'] \to \mathbb{R}^+$ is a convex function such that $\gamma(0) = 0$ and*

$$\beta(t) = \lfloor\frac{t}{t'}\rfloor\gamma(t') + \gamma(t - \lfloor\frac{t}{t'}\rfloor \cdot t')$$

then $\beta(t)$ is a service curve.

Proof: Consider the two requirements given in Definition 1 $\beta(\cdot)$ must meet to be a service curve. By substitution, $\beta(0) = 0$ which satisfies condition 1. For condition 2, let $t = q_1 \cdot t' + r_1$ and $s = q_2 \cdot t' + r_2$ such that $r_1, r_2 \in [0, t')$ and $q_1, q_2 \in \mathbb{Z}^+ \cup \{0\}$. Then:

$$\beta(t) = q_1\gamma(t') + \gamma(r_1) \tag{2}$$

$$\beta(s) = q_2\gamma(t') + \gamma(r_2) \tag{3}$$

Case I: $r_1 + r_2 \le t'$:

$$\beta(t + s) = (q_1 + q_2)\gamma(t') + \gamma(r_1 + r_2) \tag{4}$$

If $r_1 = r_2 = 0 = r_1+r_2$, then $\beta(r_1)+\beta(r_2) \le \beta(r_1+r_2)$ holds. If $r_1 + r_2 > 0$, using Jensen’s inequality, we have:

$$\frac{r_1}{r_1 + r_2}\gamma(r_1 + r_2) + \frac{r_2}{r_1 + r_2}\gamma(0) \ge \gamma(r_1) \tag{5}$$

$$\frac{r_2}{r_1 + r_2}\gamma(r_1 + r_2) + \frac{r_1}{r_1 + r_2}\gamma(0) \ge \gamma(r_2) \tag{6}$$

Summing together (5) and (6):

$$\gamma(r_1 + r_2) \ge \gamma(r_1) + \gamma(r_2) \tag{7}$$

Using equation (7) with (2),(3),(4), we have:

$$\beta(t + s) \ge \beta(t) + \beta(s)$$

Case II: $r_1 + r_2 > t'$: Let $\tau = r_1 + r_2 - t'$:

$$\begin{aligned}\beta(t + s) &= (q_1 + q_2 + 1)\gamma(t') + \gamma(\tau)\\ &= (q_1 + q_2)\gamma(t') + \gamma(t') + \gamma(\tau)\end{aligned} \tag{8}$$

We know $r_1 < t'$ and $r_2 < t'$. Hence, $\max(r_1, r_2) < t'$. Since, $\tau + t' = r_1 + r_2$ and $r_1 + r_2 > t'$, we have $0 < \tau < \min(r_1, r_2)$. Hence, $t' - \tau > 0$. Using Jensen’s inequality, we have:

$$\left(\frac{t' - r_1}{t' - \tau}\right) \cdot \gamma(\tau) + \left(1 - \frac{t' - r_1}{t' - \tau}\right) \cdot \gamma(t') \ge \gamma(r_1) \tag{9}$$

$$\left(\frac{t' - r_2}{t' - \tau}\right) \cdot \gamma(\tau) + \left(1 - \frac{t' - r_2}{t' - \tau}\right) \cdot \gamma(t') \ge \gamma(r_2) \tag{10}$$

Summing equations (9) and (10), we have:

$$\gamma(t') + \gamma(\tau) \ge \gamma(r_1) + \gamma(r_2) \tag{11}$$

In this case as well, using equation (11) with (2), (3), (8), we have:

$$\beta(t + s) \ge \beta(t) + \beta(s)$$

Hence, proved. ■

Theorem 3. *If control signal $u(\cdot)$* violates *special service curve $\beta(\cdot)$ for the first time at time T, then*

$$\exists\, t \in [T-t', T).\, U(t) + \beta(T-t) > U(T)$$

Proof: Violating a service curve at time T is equivalent to [6, p. 27]:

$$U(T) < \sup_{0 \le \tau < T} \{U(\tau) + \beta(T-\tau)\}$$

Let s be a point such that:

$$U(T) < U(s) + \beta(T-s) \tag{12}$$

Let $t = q \cdot t' + s$, where q is chosen such that $t \in [T-t', T)$. Using properties of *special service curves*, we have:

$$\begin{aligned} \beta(T-s) &= \beta(T-t+q\cdot t') \\ &= \beta(T-t) + q\cdot\beta(t') \end{aligned} \tag{13}$$

We know that the control signal has not violated the service curve before time T. Hence, we have:

$$\begin{aligned} U(t) - U(s) &\ge \beta(t-s) \\ &= q\cdot\beta(t') \\ \Longrightarrow\; U(s) + q\cdot\beta(t') &\le U(t) \end{aligned} \tag{14}$$

Combining equation (12) with (13) and using (14):

$$\begin{aligned} U(T) &< U(s) + \beta(T-s) \\ &= U(s) + q\cdot\beta(t') + \beta(T-t) \\ &\le U(t) + \beta(T-t) \end{aligned}$$

and since $t \in [T-t', T)$, proved. ■

Theorem 4. *Given $U(\cdot)$, an energy function, and $\beta(\cdot)$, a* special service curve *with period t', then for $t \in [t_K, t_K + t']$:*

$$\hat{U}(t) = \max \begin{cases} U(t-t') + \beta(t'), \\ U(t_I) + \beta(t-t_I), \\ U(t_{I+1}) + \beta(t-t_{I+1}), \\ \vdots \\ U(t_K) + \beta(t-t_K) \end{cases}$$

where:

- *I is chosen such that $t_I \le t - t' < t_{I+1}$, $t_i \in \mathbf{T}_U$,*
- *$t_K = \max(\mathbf{T}_U)$, and*
- *$\hat{U}(\cdot)$ is the* ideal prediction *for $U(\cdot)$ and $\beta(\cdot)$.*

Proof: Consider the function $f_t(s) = U(s)+\beta(t-s)$ defined for $s \in [t-t', t_K]$. It can be rewritten as:

$$f_t(s) = \begin{cases} U_0(s) + \beta(t-s) & \mid s \in [t-t', t_i] \\ U_1(s) + \beta(t-s) & \mid s \in [t_i, t_{i+1}] \\ \vdots & \vdots \\ U_{K-i}(s) + \beta(t-s) & \mid s \in [t_{K-1}, t_K] \end{cases}$$

where all $t_i \in \{\tau \mid \tau \in \mathbf{T}_U \wedge \tau > t\}$, and $U_j(t) = U(t), 0 \le j \le K-i$.

Note that all $U_j(s)$ are linear functions of s. We know that $\beta(t-s)$ is a convex function since $t-s \in [0, t']$. Adding a linear function to a convex function results in another convex function. Hence, $h_t^j(s) = U_j(s)+\beta(t-s)$ for $0 \le j \le K-i$ are all convex functions which attain their maxima at the boundaries.

From the definition of *ideal prediction*, we know $\hat{U}(t) = \max_s\{f_t(s)\}$. Now the maximum of the function $f_t(s)$ can be written as:

$$\hat{U}(t) = \max \begin{cases} \sup_s\{h_t^0(s)\} & \mid s \in [t-t', t_i] \\ \sup_s\{h_t^1(s)\} & \mid s \in [t_i, t_{i+1}] \\ \vdots & \vdots \\ \sup_s\{h_t^{K-i}(s)\} & \mid s \in [t_{K-1}, t_K] \end{cases}$$

By rewriting $\sup_s\{h_t^j(s)\}$ as $\max\{h_t^j(\tau_1), h_t^j(\tau_2)\}$ where τ_1, τ_2 are the end points of the j^{th} linear segment and removing duplicate terms:

$$\hat{U}(t) = \begin{cases} U(t-t') + \beta(t'), \\ U(t_i) + \beta(t-t_i), \\ \vdots \\ U(t_K) + \beta(t-t_K) \end{cases}$$

Hence, proved. ■

Theorem 5. *Given $U(\cdot)$, an energy function, and $\beta(\cdot)$, a* special service curve *with period t', then for $t \in [t_K, t_K + t']$:*

$$\hat{V}(t) = \max \begin{cases} U(t-t') + \beta(t'), \\ U(t_I) + \beta(t-t_I), \\ U(t_{I+1}) + \beta(t-t_{I+1}), \\ \vdots \\ U(t_K) + \beta(t-t_K) \end{cases}$$

where:

- *I is chosen such that $t_I \le t - t' < t_{I+1}$, $t_i \in \mathbf{T}_U$,*
- *$t_K = max(\mathbf{T}_U)$, and,*
- *$\hat{V}(\cdot)$ is the value returned after executing the algorithm 1 with $U(\cdot)$ and $\beta(\cdot)$ as input.*

Proof: In the algorithm 1, consider the loop between lines 3–10. It can be shown by induction that after $K-I$ iterations:

$$\hat{V}_{K-I}(t) = \max \begin{cases} U(t_I) + \beta(t-t_I), \\ U(t_{I+1}) + \beta(t-t_{I+1}), \\ \vdots \\ U(t_K) + \beta(t-t_K) \end{cases}$$

Finally, in the next iteration, the last linear *piece* of the piecewise linear function C is made by joining the points

$(t_{I-1}+t', U(t_{I-1})+\beta(t'))$ and *scTip* which is the point $(t_I+t', U(t_I)+\beta(t'))$ on line 6 in the algorithm. It is easy to verify that the point $(t, U(t-t')+\beta(t'))$ lies on this line segment.

Hence, we have:

$$\hat{V}_{(K-I+1)}(t) = \max \begin{cases} U(t-t')+\beta(t'), \\ U(t_I)+\beta(t-t_I), \\ U(t_{I+1})+\beta(t-t_{I+1}), \\ \vdots \\ U(t_K)+\beta(t-t_K) \end{cases}$$

Further, $\hat{V}_{K-I+1}(t)$ is not modified later, since for all subsequent iterations $\max\{\mathbf{T}_C\} < t \implies C(.)$ is not defined at t. Hence, `maxCover` will return the value $\hat{V}_{(K-I+1)}(t)$ for all following iterations.

Hence, proved. ■

Lemma 1. *Given a total energy function $U(\cdot)$ defined till time T and a special service curve $\beta(\cdot)$, for their* ideal prediction *$\hat{U}(\cdot)$ defined in Definition 5, we have:*

$$\hat{U}(t_2) - \hat{U}(t_1) \geq \beta(t_2 - t_1) \tag{15}$$

for any $t_2 \geq t_1 \geq T$, assuming that $U(.)$ satisfies the service curve constraint.

Proof: Let s_1, s_2 be:

$$s_1 = \arg \max_{t_1 - t' \leq s \leq T} \{U(s) + \beta(t_1 - s)\}$$
$$s_2 = \arg \max_{t_2 - t' \leq s \leq T} \{U(s) + \beta(t_2 - s)\} \tag{16}$$

So that:

$$\hat{U}(t_1) = U(s_1) + \beta(t_1 - s_1) \tag{17}$$
$$\hat{U}(t_2) = U(s_2) + \beta(t_2 - s_2) \tag{18}$$

Assume that equation (15) is false. Then using equations (17) and (18), we must have:

$$\begin{aligned} &\hat{U}(t_2) - \hat{U}(t_1) = \\ &U(s_2) + \beta(t_2 - s_2) - (U(s_1) + \beta(t_1 - s_1)) \\ &\qquad < \beta(t_2 - t_1) \end{aligned} \tag{19}$$

Rearranging terms in equation (19) and using the super-additive property of service curves, we have:

$$\begin{aligned} U(s_2) - U(s_1) < \quad & \beta(t_2 - t_1) + \beta(t_1 - s_1) \\ & -\beta(t_2 - s_2) \\ \implies U(s_2) - U(s_1) < \quad & \beta(t_2 - s_1) - \beta(t_2 - s_2) \end{aligned} \tag{20}$$

Equation (20) can be rewritten as:

$$U(s_2) + \beta(t_2 - s_2) < U(s_1) + \beta(t_2 - s_1) \tag{21}$$

Now consider the possible values of s_1.

Case I: $s_1 \in [t_2 - t', T]$ then equation (21) contradicts the definition of s_2 in equation (16).

Case II: $s_1 \in [t_1 - t', t_2 - t')$ Let $\tau = t_2 - t' - s_1$. Because $\beta(\cdot)$ is a *special service curve* and $t_2 - s_1 > t'$, we have:

$$\beta(t_2 - s_1) = \beta(\tau) + \beta(t')$$

Since $U(\cdot)$ satisfies the service curve and $t_2 - t' > s_1$, we have:

$$\begin{aligned} \beta(\tau) &\leq U(t_2 - t') - U(s_1) \\ \implies U(s_1) + \beta(\tau) + \beta(t') &\leq U(t_2 - t') + \beta(t') \\ \implies U(s_1) + \beta(t_2 - s_1) &\leq U(t_2 - t') + \beta(t') \end{aligned} \tag{22}$$

Combining equation (22) and (21), we get:

$$U(s_2) + \beta(t_2 - s_2) < U(t_2 - t') + \beta(t')$$

which is contradicts the definition of s_2 given in equation (21).

Hence, proved. ■

Theorem 6. *Given a total energy function $U(\cdot)$ defined for $t \leq T$ and a special service curve $\beta(\cdot)$ with period t', their* ideal prediction *function $\hat{U}(\cdot)$ defined in Definition 5 satisfies the service curve constraint.*

Proof: Assume that the service curve is violated. Hence, we have:

$$\begin{aligned} &\exists\, t \in [T, T+t'],\ s \in [0, t]. \\ &\qquad U(s) + \beta(t - s) > \hat{U}(t) \end{aligned}$$

By construction of $\hat{U}(\cdot)$, we know that such an s cannot be in the interval $[t - t', T]$. And by using Theorem 3, we can assume without loss of any generality that $s \in [t - t', t]$. By taking intersection of the two intervals, we can conclude that $s \in [T, t]$. However, by Lemma 1 we know that there are no service curve violations for $s > T$ for the *ideal prediction*. There exists no $s \in [t - t', t]$ such that $U(s) + \beta(t - s) > \hat{U}(t)$.

Hence, we arrive at a contradiction. ■

OPTIMIZED IMPLEMENTATIONS

Here, we describe the optimal implementation of `maxCover` and `limitTime` such that they take $O(N)$ time, where N is the number of segments in the *special service curve* instead of the naive implementation which may take $O(N+K)$ time.

To see why the naive implementation may take $O(N+K)$ time, consider the number of segments in $C(\cdot)$ in iteration j. First assume that the $\hat{V}(\cdot)$ is saved as a list, in which additions and *one step forward/behind* operations can be done in $O(1)$ time. Now, given that the service curve is a monotonically increasing convex function, and that the old prediction $\hat{V}_{K-j}(\cdot)$ is an increasing function, it is easy to see that $C(\cdot)$ can intersect the existing prediction at most once. This intersection can increase the number of segments in $\hat{V}_{K-j+1}(\cdot)$ by at most one. Hence, if we loop over all the elements in $C(\cdot)$ and $\hat{V}_{K-j}(\cdot)$, we will have do $O(N+j)$ operations, which will result in $O(N+K)$ worst case performance.

However, in the next iteration, at least one segment of $\hat{V}_{K-j}(\cdot)$ will become invariant for all the following iterations. Hence, we can skip over this element in the next iteration. To do this one would save the *starting segment* for `maxCover` in the next iteration beforehand. Hence, the number of operations will again remain only $O(N)$ (at most 1 more segment added by intersection of $\hat{V}_{K-j}(\cdot)$ and $C(\cdot)$ and at least 1 segment deleted because of invariance). Hence, the algorithm would remain $O(N)$ for all following iterations.

Communication protocol with Metering wrapper

Utkarsh Upadhyay

1 Introduction

This document explains the structure of the ZPlugController (Metering) module and the communication protocol between it and the rest of the world. The communication with the wrapper is not explained here. For those details, please refer to the documentation for the wrapper.

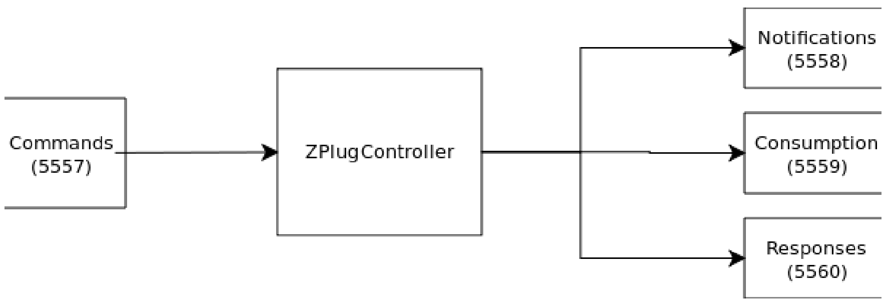


Figure 1: Communicating with the ZPlug controller (Port numbers in parentheses)

All communication happens over ZMQ sockets[1] and is done using JSON[2] strings. The COMMAND port is a ZMQ PULL port, while all the output ports are ZMQ SUB ports. Except the RESPONSE , the other two ports send information in envelopes, with the first part of the message containing the header and the other part containing the message.

Also, note that ALL strings are **case-sensitive** and have underscores instead of spaces. The general convention is that *ALL-CAPS* are used for message headers and commands to be sent to the controller. The replies are usually sent in *small-case.*

Finally, remember that the latest documentation is always the source code itself.

2 Common reply formats and notation

The **timestamp** dictionary entry will be present in all replies received from the ZPLUGCONTROLLER on all ports.

If the command sent to the ZPLUGCONTROLLER cannot be parsed at all, the reply will still be a valid JSON string. The format of the error messages sent will be:

[1]www.zeromq.org
[2]www.json.org

Response:	error
`msg`	A string explaining the error message.
`detail`	If possible, the erring portion of the message.

If the command sent contains many different commands (or a command like **TURN_OFF** with multiple arguments), then as many instructions will be carried out as possible and error messages will be returned for the rest.

3 Consumption messages

The consumption port delivers two kinds of messages, under two different headers. The first header contains the information about the current consumption taking place at the plug. These messages are sent at a regular period which is set to **5sec** by default but can be changed by sending a command to the ZPlugController .

Header:	CONSUMPTION
`nwk_addr`	Network address of the device which sent this report.
`cluster`	String description of simple metering cluster.
`cluster_id`	ID of the simple metering cluster.
`multiplier`	Multiplier.
`divisor`	Divisor.
`unit_of_measurement`	Units (after multiplication/division).
`instantaneous_demand`	Instantaneous demand (units decided after multiplication/division).

The message arriving under the other header contains information about the status of the Plug (whether they are turned on or off). Note that because the messages sent to the plugs are not guaranteed to have their effect (see Section 6), capturing these packets would be the only way of reliably confirming the status of plugs.

Header:	ON_OFF
`nwk_addr`	Network address of the device which send this report.
`cluster`	String description of on-off cluster.
`cluster_id`	ID of on-off cluster.
`on_off_status`	"ON" or "OFF"

4 Notification messages

The notification messages are sent under two headers. Note that these may be send multiple times as a device provides increasing amount of data to the UBEE controller.

These messages arrive from the ZPlugs (and possibly ZRCs). They contain information about the devices in the network.

Header:	ZNode
`type`	"UBEE"
`source`	"ZNode"
`nwk_addr`	Network address of this device.
`event`	Event ID (integer).
`description`	String describing the event.
`model_id`	Model ID of the device.
`device_type`	The type of ZigBee device.

These messages are sent by the UBEE module itself while starting/stopping and sometimes on errors.

Header:	UBEE
`type`	"UBEE"
`source`	"UBEE"
`event`	Event ID (integer).
`description`	String describing the event (UPDATE, STARTED, etc.)

5 Commands and responses

This section gives a description of the commands which can be send to the ZPlugController and what are the messages broadcast on the Response port.

The rationale behind not having any headers on the replies (unlike for messages sent on Consumption port and Device Notification port) and yet publishing the message (instead of REQ-REP pattern) is:

- They can be captured by the logger(s).
- There should be only one command sender and in the presence of two (or more) command senders, the unfiltered result should be visible to the other party as well (for alerts).

5.1 QUIT

Requests the ZPlugController to stop operation.

Command:	QUIT
const	*True*

Response:	quit
const	*True*

Also note that this response will be emitted if ZPlugController is asked to quit by interruption on the server-end.

5.2 RESTART

Restarts the ZPlugController .

Command:	RESTART
const	*True*

Response:	restart
const	*True*

5.3 TURN_OFF

Turns ZPlugs OFF.

Command:	TURN_OFF
[**nwk_addr**]	List of integers giving the network addresses which should be turned off.

Response:	turn_off
`<Each nwk_addr>`	String "OK", or if the address was not legal, with "Invalid Address".

5.4 TURN_ON

Turns ZPlugs ON.

Command:	TURN_ON
[**nwk_addr**]	List of integers giving the network addresses which should be turned on.

Response:	turn_on
`<Each nwk_addr>`	String "OK", or if the address was not legal, with "Invalid Address".

5.5 TOGGLE

Toggles ZPlugs: if they were ON, turn them OFF, and vice-versa.

Command:	TOGGLE
[**nwk_addr**]	List of integers giving the network addresses which should be toggled.

Response:	toggle
`<Each nwk_addr>`	String "OK", or if the address was not legal, with "Invalid Address".

5.6 UPDATE_PERIOD

Updates the timeout for messages on the Consumption messages.

Command:	UPDATE_PERIOD
const	New value of timeout **(in microseconds)**

Response:	update_period
`new_value`	The new period.
`old_value`	The old period.

5.7 PERIOD

Fetches the update period for the Consumption messages.

Command:	PERIOD
const	*True*

Response:	period
const	An unsigned integer giving the current period of messages.

5.8 VERSION

Request the version of ZMQ library being used for communication. As per the ZMQ guide, a version mismatch is the most common source of error in case messaging is not working.

Command:	VERSION
const	*True*

Response:	version
const	The ZMQ version in the format "major.minor.patch"

6 A note on reliability

It should be noted here that this module does not make any guarantees on the state of the underlying ZPlugs after commands are executed on them. For example, it may happen that even after executing an OFF command, the status of the ZPlug remains ON. This (unchanged) status will be reflected in the messages received from the CONSUMPTION port though. Hence, each layer needs to ensure that the messages are indeed acted upon.

However, automatically resending messages to the ZPLUGCONTROLLER is STRONGLY DISCOURAGED. This can lead to race conditions among the applications. Currently, there are no clear apparent solutions to this problem.

Communication API of SHC

Utkarsh Upadhyay

February 3, 2012

1 Introduction

This document explains the structure of the SHC module and the communication protocol between it and the UI, algorithm and modules. The communication with the wrapper is not explained here. For those details, please refer to the documentation for the wrapper.

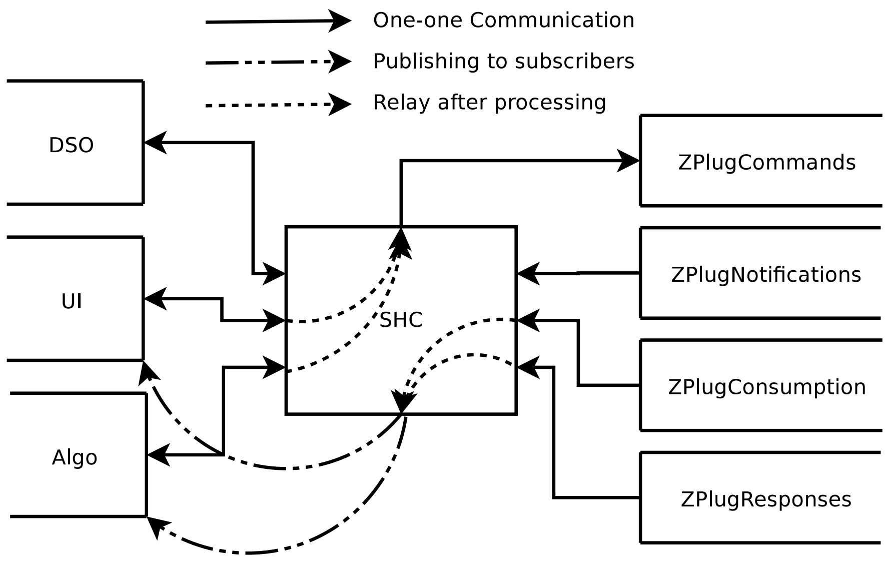


Figure 1: Simplified schema of communication with the SHC

All communication happens over ZMQ sockets[1] and is done using JSON[2] strings. The one-one communication happens by opening a ZMQ PULL port and the responses are sent via a dedicated PUB port. These ports are kept separate instead of made common to allow for easy port to a multi-threaded application.

The ports which publish messages are ZMQ SUB ports. Note that only one of those is shown in the Figure 1 while there are multiple such ports, e.g. for sending error messages and heartbeats. More details can be found in the configuration file.

[1]www.zeromq.org
[2]www.json.org

Also, note that ALL strings are **case-sensitive** and have underscores instead of spaces. The general convention is that *ALL-CAPS* are used for message headers and commands to be sent to the controller. The replies are usually sent in *small-case*. The exceptions, if any, will be documented below.

Finally, remember that the latest documentation is always the source code itself.

2 Common reply formats and notation

The **timestamp** dictionary entry will be present in all replies received from the SHC on all ports.

If the command sent to the SHC cannot be parsed at all, the reply will still be a valid JSON string. The format of the error messages sent will be:

Response:	error
`msg`	A string explaining the error message.
`detail`	If possible, the erring portion of the message.

If the command sent contains many different commands (or a command like **TURN_OFF** with multiple arguments), then as many instructions will be carried out as possible and other uninterpreted commands will be ignored.

3 Published messages

The consumption port delivers only one kind of message which essentially contains the entire information about the state of the system:

```
{
  "CONSUMPTION": {
    "18557": {
      "instantaneous_demand": 0.0,
      "on_off_status": 1,
      "priority": "*"
    },
    "32629": {
      "instantaneous_demand": 0.035,
      "on_off_status": 1,
      "priority": "1"
    }
  },
  "SERVICE_CURVE": {
    "curve": [
      { "slope": 0.1, "length": 10000 },
      { "slope": 10.0, "length": 20000 },
      { "slope": 11.0, "length": 30000 }
    ],
    "period": 60000
  },
  "DSO": { "limit": 1.0, "guaranteed_for": 0 },
  "timestamp": 1327029098.475662
}
```

Because the port sends only one form of message, there is no header sent. However, this may change in future. Now explaining the different keys in the dictionary:

- **CONSUMPTION**: Contains a dictionary with the Network addresses (unique) of the devices as key and contains information about each device.
  1. *instantaneous_demand*: The instantaneous demand at the device in kW.
  2. *on_off_status*: Whether the device is on or off.
  3. *priority*: "*" if the device is *user controlled*, a positive integer giving the priority of the device if the device is controlled by the algorithm module. *A device with lower priority value would be turned off only after no higher priority devices are present. Higher number corresponds to lower importance.*
- **SERVICE_CURVE**: We see the entire service curve, the representation of which has been talked about in detail in the report. The representation is exactly as described.
- **DSO_CURVE**: This gives details about the DSO's latest signal.
  1. *limit*: The maximum value which can be drawn from the DSO
  2. *guatanteed_for*: The minimum number of seconds for which the signal is guaranteed to remain the same or improve. This is in line with the recommendations made earlier.

Note that because the messages sent to the devices are not guaranteed to have their effect (see Section 5), capturing these packets would be the only way of reliably confirming the status of devices.

Header:	ON_OFF
`nwk_addr`	Network address of the device which send this report.
`cluster`	String description of on-off cluster.
`cluster_id`	ID of on-off cluster.
`on_off_status`	"ON" or "OFF"

4 Commands and responses

This section gives a description of the commands which can be send to the SHC and what are the messages published in response.

The rationale for not having any headers on the replies and yet publishing the message (instead of REQ-REP pattern) is that:

- They can be captured by the logger(s).
- There should be only one command sender and in the presence of two (or more) command senders, the unfiltered result should be visible to the other party as well (for alerts).

4.1 UI commands

4.1.1 QUIT

Requests the SHC to stop operation.

Command:	QUIT
const	*True*

Response:	quit
const	*True*

4.1.2 RESTART

Restarts the SHC.

Command:	RESTART
const	*True*

Response:	restart
const	*True*

This is done by simulating a software crash, and allowing the wrapper to restart the SHC.

4.1.3 TURN_OFF

Turns devices OFF. The priority of the devices is reset to "*", or to user mode.

Command:	TURN_OFF
[**nwk_addr**]	List of integers giving the network addresses which should be turned off.

Response:	turn_off
`<Each nwk_addr>`	String "OK", or if the address was not legal, with "Invalid Address".

4.1.4 TURN_ON

Turns devices ON. The priority of the devices is reset to "*", or to user mode.

Command:	TURN_ON
[**nwk_addr**]	List of integers giving the network addresses which should be turned on.

Response:	turn_on
`<Each nwk_addr>`	String "OK", or if the address was not legal, with "Invalid Address".

4.1.5 TOGGLE

Toggles devices: if they were ON, turn them OFF, and vice-versa. The priority of the devices is reset to "*", or to user mode.

Command:	TOGGLE
[**nwk_addr**]	List of integers giving the network addresses which should be turned off.

Response:	toggle
`<Each nwk_addr>`	String "OK", or if the address was not legal, with "Invalid Address".

4.1.6 VERSION

Request the version of ZMQ library being used for communication. As per the ZMQ guide, a version mismatch is the most common source of error in case messaging is not working.

Command:	VERSION
const	*True*

Response:	version
const	The ZMQ version in the format "major.minor.patch"

4.1.7 ID

A user can request for an unique ID to identify himself which he can attach to his commands.

Command:	ID
`NAME`	Name of the user.

Response:	id
`name`	Name of this user.
`uuid`	A UUID for this user.

4.2 Algorithm Module

These are the commands which can be sent by the Algorithm module.
Note: The replies here conflict with replies of some UI module's replies (See 4.1.4 and 4.1.3).

4.2.1 ALGO_OFF

Turns devices OFF. The priority of devices remains unaltered.

Command:	ALGO_OFF
[**nwk_addr**]	List of integers giving the network addresses which should be turned off.

Response:	turn_off
`<Each nwk_addr>`	String "OK", or if the address was not legal, with "Invalid Address".

4.2.2 ALGO_ON

Turns devices ON. The priority of devices remains unaltered.

Command:	ALGO_ON
[**nwk_addr**]	List of integers giving the network addresses which should be turned on.

Response:	turn_on
`<Each nwk_addr>`	String "OK", or if the address was not legal, with "Invalid Address".

4.3 DSO's commands

These are the commands which can be sent by the DSO.

4.3.1 DSO

This changes the control signal.

Command:	DSO
`CHANGE_TO`	The next control signal
`FOR_ATLEAST`	The period for which the DSO promised to not change its signal

Response:	
`old_value`	The old control signal
`old_period`	The old guaranteed period
`new_value`	The new control signal
`new_period`	The new guaranteed period
`minimum_needed`	The minimum amount of power needed (in kW) of User mode devices
`warning`	True if the minimum consumption cannot be met with this demand
`violation`	True if the DSO *lowered* the control signal before the time it had previously promised

4.3.2 SERVICE_CURVE

This changes the service curve.

Command:	SERVICE_CURVE
[**"slope"** : ..., **"length"** : ...]	Array of (slope, length) dictionaries.

To see how this represents the service curve, refer to the report.

Command:	service_curve
const	String "OK", or some fields were not correct, a *missing field* error.

5 A note on reliability

It should be noted here that this module does not make any guarantees on the state of the underlying devices after commands are executed on them. For example, it may happen that even after executing an OFF command, the status of the underlying device remains ON. This (unchanged) status will be reflected in the updates published though. Hence, each layer needs to ensure that the messages are indeed acted upon.

However, automatically resending messages to the SHC is STRONGLY DISCOURAGED. This can lead to race conditions among the applications. Currently, there are no clean solutions to this problem.