

ホームネットワークシステムの概要と現状 (続編)

報告書

平成 28 年 3 月

一般財団法人テレコム先端技術研究支援センター

調査委員会名簿

(順不同 敬称略)

委員長	丹 康雄	北陸先端科学技術大学院大学 情報科学研究科 教授
事務局	門馬 弘	一般財団法人テレコム先端技術研究支援センター 研究企画部 部長
事務局	牧野 淳一	一般財団法人テレコム先端技術研究支援センター 事業部 調査役

目 次

第1章 ホームネットワークシステムにおける共通インタフェース	1
1.1 様々なインタフェース	1
1.2 端末機器接続のインタフェース	6
1.3 宅内 IP ネットワークインタフェース	14
1.4 サービスプラットフォーム-ホームゲートウェイインタフェースとサービスプラットフォーム-サービスインタフェース	17
1.5 API の抽象度	20
1.6 サービスプラットフォーム型に向けての道筋	25
第2章 ホームネットワークのセキュリティ	27
2.1 情報セキュリティからサイバーセキュリティへ	27
2.2 IOTセキュリティ	31
2.3 今後に向けて	38
第3章 オープンソース ECHONET Lite 開発環境 echowand	45
3.1 echowand と本章の位置づけ	45
3.2 echowand	46
3.2.1 全体構成	46
3.2.1.1 common モジュール	48
3.2.1.2 info モジュール	48
3.2.1.3 subnet モジュール	48
3.2.1.4 transaction モジュール	49
3.2.1.5 object モジュール	53
3.2.1.6 service モジュール	55
3.2.2 拡張ライブラリとアプリケーション	55
3.2.3 機器オブジェクト作成方法	57
3.3 仮想オブジェクト生成	58
3.3.1 実行環境	58
3.3.1.1 プログラム構成	59
3.3.1.2 プログラム配置	60
3.3.2 実行方法	61
3.3.3 機器設定 JavaScript プログラム記述	62

3.3.3.1	ECHONET Lite オブジェクト記述用ヘルパーオブジェクト	62
3.3.3.2	ECHONET Lite オブジェクトの設定記述	63
3.3.3.3	ECHONET Lite オブジェクト動作のカスタマイズ	65
3.3.3.4	動作カスタマイズ例	69
3.3.4	ECHONET Lite オブジェクトの登録	71
3.3.5	実行例	72
3.4	まとめ	72
付録 A	機器設定 JavaScript プログラムサンプル	73

第1章 ホームネットワークシステムにおける共通インタフェース

昨年度報告書の第1章に述べたように、ホームネットワークシステムに期待されるサービスには様々な種類があり、それぞれ技術的な要求要件が異なっている。また、サービスを実現していくためには具体的なアプリケーションを実現していくため、昨年度報告書の第2章に述べたように、5つの要素技術が必要となる。この中には、伝送メディアのようにアプリケーションの要求に特有の性質、例えば、AV(Audio Visual)アプリケーションで必要とされる DRM(Digital Right Management: コンテンツ保護のための機能)のようなものも含まれるが、センシングし、その結果に基づいた判断を行なってアクチュエーションをし、それらの一連の記録をデータベースに蓄積するといった部分はどのようなアプリケーションでも枠組みとして共通である。また、センサ情報でも部屋の温度や明るさといった基礎的ものは極めて多くのアプリケーションで共通であり、これらに変更を加えるための空調や照明といった装置(アクチュエータ)も共通であると考えられる。従って、共通であるのみなせるものをできるだけまとめて実装すれば流用が可能となり、様々な意味で経済性に優れる事となる。本章では、こうした共通化の観点で幾つかの論点について述べる。

1.1 様々なインタフェース

図 1.1 に、昨年度報告書第2章で説明した制御およびデータベース化のためのシステム形態のあり方を示す。センサや外部からの情報を収集し、そこに過去の実績からの情報も加えて判断を行ない、アクチュエータを動作させるためには、全ての機能が家庭内のコントローラに存在するようなスタンドアロン型も考えうるが、コントローラの開発効率や運用コストを考えればクラウドを活用する形が有利であり、特に複数のサービス間で連携をはかったり、機器を共有したりするためにはプラットフォーム型に発展していくのが自然であるという論点で説明に使った図である。

ホームネットワークシステムの場合、ICT(情報通信)システムとは無関係に建物として一軒の家があり、家族の生活があるところから出発しているという経緯を鑑みればユーザー

が中心にあり、その周囲に場としての家庭環境があって、そこに各種のホームネットワークサービスを提供するという構図になることは容易に理解できよう。このような利用形態のシステムであれば、複数のサービスが連携して一体感のある形で提供されるのが当然であり、各機器がバラバラにクラウド連携し、それぞれのアプリケーションごとに温度センサや照度センサを設置するというよりはシステムとして共有したセンサ類が設置されている方が自然である。実のところ、IoT システム一般についていえば、必ずしも異なるサービスで同一のプラットフォームを利用することが自然であるとまでは言えず、ホームネットワークシステムという例は、サービス間連携が強く望まれる特別な例という見方もできる。

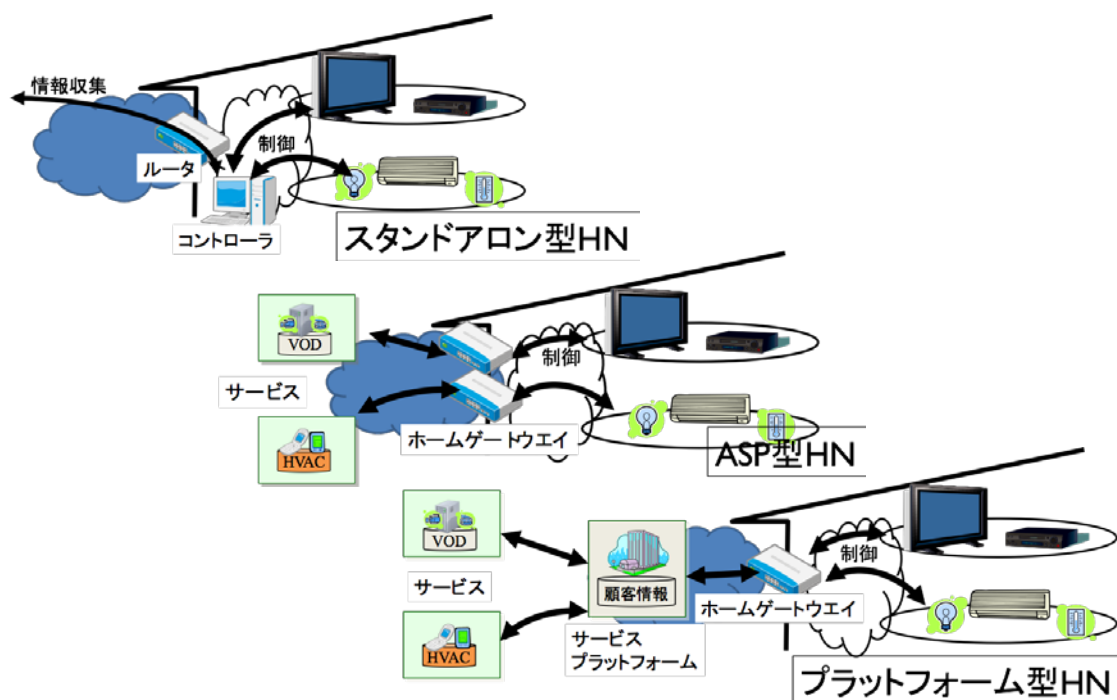


図 1.1 サービス実現のための三形態

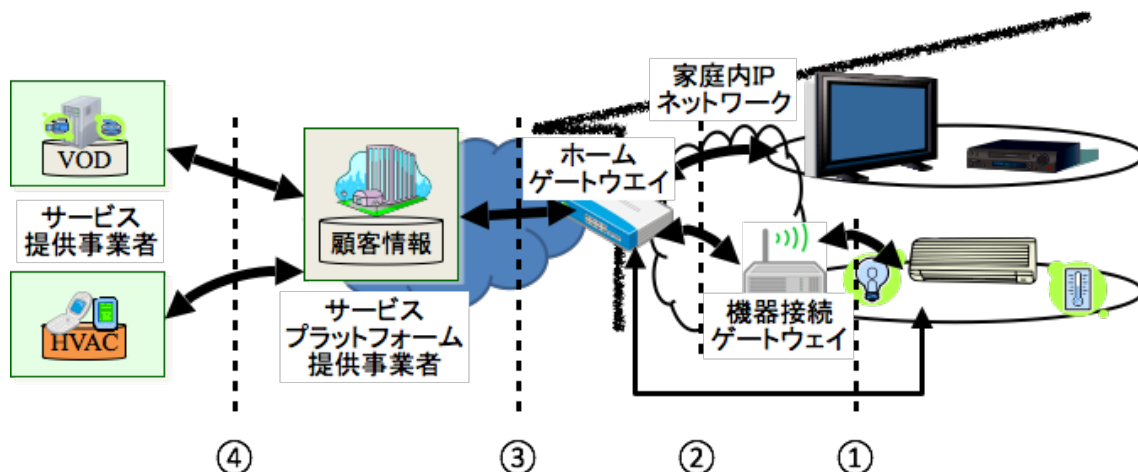


図 1.2 様々なインタフェース

上記の三形態の中でも最も登場するアクターの多いサービスプラットフォーム型を基に、どのようなインタフェースがありうるか、代表的なものを示したものが図 1.2 である。

図において①から④までのインタフェースは以下の様な意味合いを有する。

①機器接続インタフェース

センシングやアクチュエーションを行なう端末機器がネットワークにつながる末端の接続インタフェース。直接②のインタフェースに接続する、つまり、直接 IP ネットワークに接続するような端末機器も存在するが、多くの場合、コストの観点から端末機器は組み込みシステム用のネットワークインタフェースを有し、何らかの通信技術ヘブリッジあるいはゲートウェイとなる機器接続ゲートウェイを経由して家庭内 IP ネットワークに接続される。

また、機器接続ゲートウェイ経由ではなく家庭内 IP ネットワークを経由せずに直接ホームゲートウェイに接続する形態もとらうる。ホームゲートウェイが専用のインタフェースを持っていたり、USB のような汎用のインタフェースにインタフェース変換機器（いわゆる USB ドングル）を接続するような形態である。これは、RS-485 や RS-232C のようなシリア

ルインタフェースや、接点端子のような形であったり、ZigBee や Z-Wave のような無線通信インタフェースであったりする。何れにしても、①のインタフェースは IP レイヤ(ネットワークレイヤ)での外部との接続性がないということがポイントである。このインタフェースで接続された端末機器の情報は、一度アプリケーションまで持ち上げられ、アプリケーションが必要に応じて利用したり他のサービスと通信したりする形をとる。

家庭内 IP ネットワーク内において無線の Wi-Fi を利用している場合、インターネットと接続される有線の Ethernet からのブリッジ(あるいはルーター)の役割を果たす無線 LAN アクセスポイントが存在することになり、一見このインタフェースと類似した接続形態に見えるが、この場合には端末機器は IP レイヤにおいて直接ホームゲートウェイ(ひいてはインターネット)に到達可能な家庭内 IP ネットワークの一員となっており、この接続インタフェースとは見なされない。逆に、端末機器が直接 Ethernet の端子を持っていたり、家庭内 IP ネットワークとは異なるプライベートなネットワークに接続され、機器接続ゲートウェイを経由して初めて家庭内 IP ネットワークに接続できるような場合には、この①の接続インタフェースであるとみなされる。これらについては 1.2 で後述する。

①のインタフェースにおいては、端末機器が実現するサービスに特化した様々な特殊な機能を実現しているものが多い。たとえば、映像コンテンツを伝送する技術では単に高速な通信が行えるだけでなく、コンテンツの権利保護を実現する機能が実装されている必要があるし、センサを接続する無線インタフェースの場合には、電池駆動するセンサの電源消費を抑えるため、通常はスリープし間欠的にしか通信しないような技術が必要とされる。このような事情から、機器接続ゲートウェイでは単にプロトコルを変換するというだけでなく、一旦機器接続ブリッジ/ゲートウェイで通信を終端させアプリケーションなど上位レイヤの処理を経てから改めて別の通信として家庭内 IP ネットワークに送られることになる。

②宅内 IP ネットワークインタフェース

家庭内においてインターネットプロトコル(IP)を用いて通信されるネットワークに接続するインタフェースである。この IP ネットワークはホームゲートウェイを経由して外部のインターネットと接続されており、NAT(Network Address Translation)を受けることがあるにせよ、ネットワークレイヤで直接インターネットと接続されることになる。これが①のインタフェースとの相違である。

家庭内 IP ネットワーク内には無線 LAN、PLC(Power Line Communication:電力線通信)、HDMI の Ethernet チャンネルなど、様々な形態の物理レイヤやデータリンクレイヤを有する技術が使われるが、この上に載る IP パケットがホームゲートウェイ経由でルーティングされるものについては、同じ Ethernet スイッチに接続されているものと同一とみなせるものとし、全てこのインタフェースに接続されているものとみなされる。これは、ホームネットワークにおいてはネットワークレイヤとしては単一のブロードキャストドメインを形成している家庭内 IP ネットワークは、物理レイヤやデータリンクレイヤで見ると様々な技術をブリッジ接続して構成されている、ということを意味している。この状況は一般ユーザーのみならず専門家にとってもわかりにくいものであり、トラブルを引き起こす元となる。この問題を解決するためには物理レイヤやデータリンクレイヤにおけるネットワーク(リンク)トポロジを把握できるためのしくみが重要であり、ネットワーク管理の問題として第 3 章で後述する。

③サービスプラットフォームーホームゲートウェイインタフェース

家庭内に設置されているホームゲートウェイとインターネット内のサービスとを接続するインタフェースであるが、ここではサービスプラットフォームの存在を前提としているため、サービスプラットフォームとホームゲートウェイの間のインタフェースということになる。

このインタフェースの意味合いは後述のように、どのような役割をホームゲートウェイが果たすかに依存するが、典型的なサービスプラットフォーム型の構成においては、サー

ビスプラットフォーム提供事業者として必要な API(Application Programming Interface)を提供するために、家庭内ネットワークにおいて直接行わねばならない操作を実現するための情報を交換するインタフェースとなる。こうした形態の場合、③のインタフェースは標準化されたオープンなものというよりは、サービスプラットフォーム提供事業者の内部仕様となることが想定される。つまり、④はサービス提供事業者へ提供する API としての外部インタフェースで、②は家庭内の機器を制御したり情報を取得したりするため何れも外部インタフェースであり、他社の製品やサービスとの接続を想定しているのに対して、③はサービスプラットフォーム提供事業者の内部規定として実現されるという意味である。

もちろん、サービスプラットフォームとホームゲートウェイの提供者が異なり、③が外部インタフェースとして実装されるような形態も考えうるが、その場合にはホームゲートウェイレベルで標準化すべき項目が極めて多数に及び、相互接続性を確保することも容易ではないことから、あまり早期に実現できる形態とは考えにくく、当面は③は内部インタフェースと考えるのが妥当であると思われる。

④サービスプラットフォーム-サービスインタフェース

サービスプラットフォーム提供事業者がユーザー向けサービスを実装する事業者に向けて提供する API となる。この位置づけは、サービス提供事業者、サービスプラットフォーム提供事業者、ホームゲートウェイの役割分担に依存して大きく変わることになる。これについては、続く節で述べる。

以下の節において、各インタフェースについて述べていくことにする。

1.2 端末機器接続のインタフェース

1.1 の①で示した端末機器接続インタフェースは端末機器側が持つネットワークインタ

フェースそのものである。端末機器の種類は様々であり、HDD レコーダやエアコンのように単価が 10 万円を超えるような家電製品も珍しくないが、逆に温度センサや人感センサのように、価格的にも数百円程度ながら一つの家庭の中に数個から数百個配置することが望まれるようなものも存在する。

これらの機器を接続するにあたって、考慮しなければならない点が大きく分けて二点ある。一点目は言うまでもなくコストである。本体機能を実現するよりもはるかに高いマイコンを必要としてしまうようなネットワークプロトコルで接続するのは現実的ではない。前述したように各端末機器に必要とされる固有の機能や性質というものがあり、これを全て汎用のネットワーク技術で実現するのは極めて高くついてしまう。テレビとレコーダの間の HDMI では 10Gbps に及ぶ速度での通信が行われているが、10Gbps の Ethernet インタフェースはパソコンに装着するカードの形で数万円以上しており、組込みで実装したとしてもコスト的に見合うものではない。組込み用の低廉かつ必要な機能や性能を有するリンク技術が必要となるわけである。

二点目は、機器の利用のしかたに関するものである。パソコンや HDD レコーダはスマートフォンやテレビなど様々な機器からアクセスができてコンテンツの共有などが図れるのが望ましいが、太陽電池や蓄電池、燃料電池といったエネルギー機器については、その運転方法は一般ユーザーが直接行なうべきものではなく、むしろ、ネットワーク内でそうした端末が見えていることは事故を招く恐れもある。また、悪意のある第三者がエアコンや電子レンジなどの機器を不法に操作し、人体に危害を加えるような状況になることも考えられる。こうした状況を鑑みれば、ホームネットワーク内の一部の機器類はパソコンなどがつながっている家庭内の主たる IP ネットワークからは適切に分離されている方が望ましいといえる。

実際のところ、HEMS(Home Energy Management System:家庭用エネルギー管理システム)の場合には HEMS 関連機器は①の点で接続される独立したネットワークを形成し、これらの機器を制御する HEMS コントローラのみが②の点に接続されるような形態で使われ

ることがほとんどであった。特に注目すべきは、HEMS 関連機器のネットワークインタフェースは近年ではほとんどが IP(インターネットプロトコル)に基づくものであり、技術的には全て②の点で接続することが可能であるという点である。つまり、この①のインタフェースにつながる機器は IP 接続ができないためにゲートウェイを介しているのではなく、意図的に分離するのが目的となっているわけである。

この状況は、TTC(一般社団法人情報通信技術委員会)で制定されたテクニカルレポート「TR-1043 ホームネットワーク通信インタフェース実装ガイドライン」[1]においても明示されている。TR-1043 は、制御系のホームネットワークを対象に、特に ECHONET Lite 規格を上位レイヤとして実装することを想定した各種物理レイヤおよびデータリンクレイヤの実装方式を記述したもので、経済産業省の HEMS 補助金においても、この TR-1043 に記載されているものの中から推奨規格として補助金対象となりうる技術が指定されていた。つまり、TR-1043 に記載されていない技術は補助金の対象にもなり得ないことから、実用化される可能性のあるものについては広く収録した文書となっている。また、この文書自体は必ずしも仕様が無償公開されているものだけを対象としたものではなかったが、補助金の対象になるためには技術仕様が誰でも無償で入手可能(然るべき Web サイトから誰でも pdf でダウンロードできるなど)であることが望ましいとしていたこともあり、実質的には多くの規格が TR-1043 への記載時期に合わせて自らの仕様文書を何らかの形で公開するに至っている。このように、TR-1043 はホームネットワークにおいて一定の影響力を有する文書となっている。

TR-1043 の第 3 章としてネットワークモデルと物理構成例という記載がある。内情を言えば本章は TTC 側での議論で必要性が取り沙汰されたものというよりは、経済産業省側のスマートハウス標準化検討会から、より具体的な接続のしかたに関する記述を求められて追加したものである。通信規格上の規定だけでなく、どのようにそれを使うのが適切であるかについても指針が求められているということであろう。

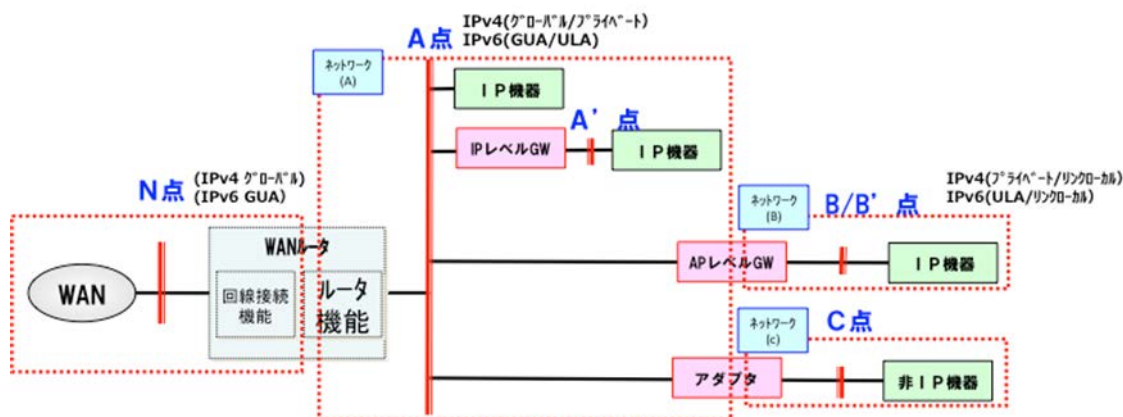


図 1.3 TTC TR-1043 にみるネットワークモデル

図 1.3 に、TR-1043 第 3 章に記載されているネットワークモデル図を示す。これは、主にネットワークレイヤまでの観点でみた接続点のモデルを表している。類似したモデルは TTC の他の文書や ITU-T の勧告文書の中にも存在するがこの図で示したモデルは他のどれとも異なったものとなっている。既存のモデルでは何れも 6LoWPAN に関する明示的な記述がなく、HEMS においては 6LoWPAN の位置づけを明らかにする必要があったため、新たに参照点モデルを作成することとした。

図中定義されている各接続点について説明する。

N 点： 広域ネットワークとホームネットワークの接続点で、通常は家庭のインターネット接続をするルーターの広域網側のネットワークインタフェースとなる。ここはインターネットサービスプロバイダや光ファイバ接続などを提供する通信事業者との切り分け点に相当する。

A 点： ホームルーターの家庭内側のインタフェースを起点として、パソコンやテレビなど、様々な家庭内の IP 接続機器を接続する点である。我が国でのホームネットワークにおいては家庭内でルーティングすることは極めて稀で、家庭内は一つのブロードキャストド

メイン(サブネット)となるのが通例である。従って、広域接続の役割を果たしているルーター以外のネットワーク機器は何れもブリッジの役割を果たすことになる。典型的な A 点の構成は、ルーターから Ethernet スイッチ(スイッチングハブ)に接続され、スイッチングハブに各機器が接続される、というものである。実際にはこれに、PLC や無線 LAN といった異種データリンク技術を用いるためのモデムやアクセスポイントが用いられることになる。これについては、家庭内 IP ネットワークということで図 1.2 の②に相当する接続点となるため、次の節で改めて取り上げる。

B 点: IP 接続ながら、WAN への到達性を持たない機器同士の接続点である。ただし、この B 点に接続されている機器のうち、少なくとも一台が A 点へのインタフェースも有し、この機器が B 点で接続されている機器と A 点に接続されている機器やインターネット内のサービスとの橋渡しをする。図中でも AP レベル GW(アプリケーションレイヤでのゲートウェイの意味)と書かれているように、あくまでもネットワークレイヤでのルーティングは行わず、B 点の端末機器との通信を一度終端させ、アプリケーションとして受け取って処理してから改めて必要に応じて A 点側の通信を行なう形となる。この場合、A 点にあるパソコンやスマートフォンなどからは B 点機器自体はネットワーク的に到達できず、AP レベル GW のみが見えているということになる。

C 点: 端末機器が IP 接続能力を持たない場合である。何らかのネットワークアダプタを経由して、A 点に接続するが、この端末機器とアダプタは必ずしも一対一の関係、つまり、端末機器がアダプタ込みで A 点接続の IP 機器になるという形を意味しているわけではなく、非 IP のプロトコルを用いて構成されているネットワークと IP 接続する A 点とをつなぐ役割をアダプタが果たしている場合もある。この場合には B 点における AP レベル GW と役割が似てくるが、AP レベル GW が必ず通信を一旦終端させるアプリケーションレイヤでの動作を伴うものであったのに対し、C 点のアダプタは非 IP ネットワークプロトコルと

IP との間を変換するブリッジのようなものである場合も考えられる。これは Z-Wave や ZigBee のような無線メッシュネットワークの場合もあるし、RS-485 や RS-422 のようなシリアルインタフェースでカスケード接続された機器群である場合もある。結局のところアダプタがどこまでの処理を行なうのかの問題であり、そこから先の通信相手との役割分担の問題でもある。

A'点と B'点: A 点ネットワークおよび B 点ネットワークにおける 6LoWPAN 機器のインタフェースである。6LoWPAN は、本来は IP パケットを載せるのが難しい低速なデータリンク技術の上に IP レイヤとの整合性を実現するためのデータリンクレイヤとネットワークレイヤの間の中間レイヤであり、IEEE802.15.4 の無線や、ITU-T G.9903(G3-PLC)の PLC といった、低速ながら消費電力が少なく通信インタフェースの価格も抑えることのできるデータリンク技術と組み合わせて使われるものである。6LoWPAN を実装することでこれらのデータリンク技術においても IPv6 による接続が可能となるが、通常の IPv6 とは直接通信はできず、IPv6 と 6LoWPAN の間のゲートウェイを必要とする。このゲートウェイはアプリケーションレイヤまでの処理は必要としない、ネットワークレイヤまでの処理を行なう中継装置となるため、図 1.3 では IP レベル GW という表現が使われている。この図では B'点に関する IP レベルゲートウェイが省略されているが、A 点ネットワーク同様に、B 点においてもゲートウェイを介して B'点ネットワークが接続されることになる。

このような定義にもとづき、本節の冒頭に述べたような、直接 IP 接続しない機器(B 点接続の機器)のある構成の例が 2 例、TTC TR-1043 には記載されている。

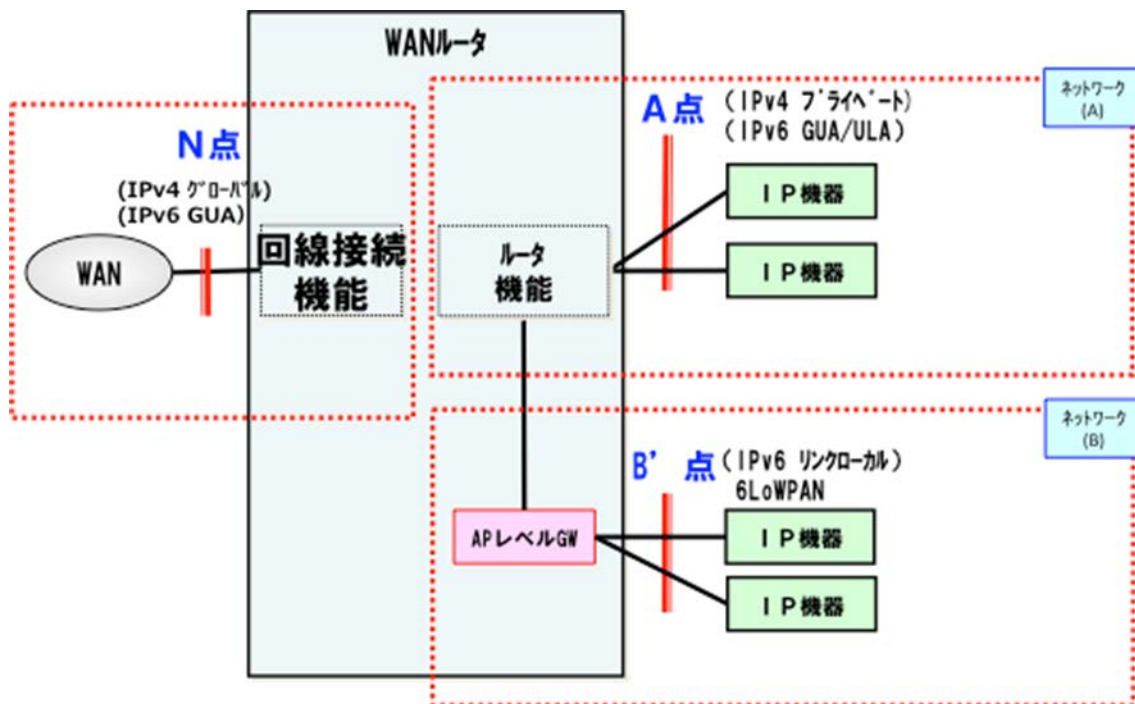


図 1.4 TTC TR-1043 にみるネットワークの構成例 1

図 1.4 に示した構成例は、様々な機能がホームゲートウェイに集約されているような例に相当している。HEMS のようなシステムを構成するとき、本来であればセンサやアクチュエータになる端末機器とそれらを制御するコントローラとが存在し、それらの間で何らかの形のネットワークを構築することになるが、多くのホームゲートウェイにはソフトウェア的に拡張性があり、この中に HEMS コントローラの機能をソフトウェアとして実装してしまうことも可能である。図 1.4 の場合、通常のインターネットサービスとしてユーザーに提供する A 点ネットワーク用の Ethernet インタフェースとは別に、HEMS コントローラの機能をホームゲートウェイ内にソフトウェアで実装するとともに USB インタフェースなどで例えば IEEE802.15.4 のインタフェースをハードとして追加し、その上に Wi-SUN の ECHONET Lite 用プロトコルスイートを実装して HEMS に関する端末機器類を接続するような形を想定している。HEMS の端末機器類は図中でも B'点になっているのでわかる

ように、6LoWPAN を用いた IPv6 での接続が行われているが、これらの機器は直接インターネットや A 点ネットワークにある他の IP 機器とは接続されず、ホームゲートウェイ内の AP レベル GW のみに接続されることとなる。この例では AP レベル GW も 6LoWPAN を用いているため、IP レベル GW は存在しない形になっている。このホームゲートウェイ内のソフトウェア実装 HEMS コントローラが機器を制御するアプリケーションの機能を持つとともに、外部のクラウドなどとの通信を、今度は A 点機器のような振る舞いで行なうため、ホームゲートウェイの中でルータ機能と接続されている。

この例では HEMS 機器が 6LoWPAN を使うネットワークで接続されたため B'点となったが、非 IP のデータリンク技術で構築されていれば C 点となる。何れの場合においても HEMS コントローラ配下のみに接続されていて A 点ネットワークの機器からの直接接続がないという点がポイントとなる。

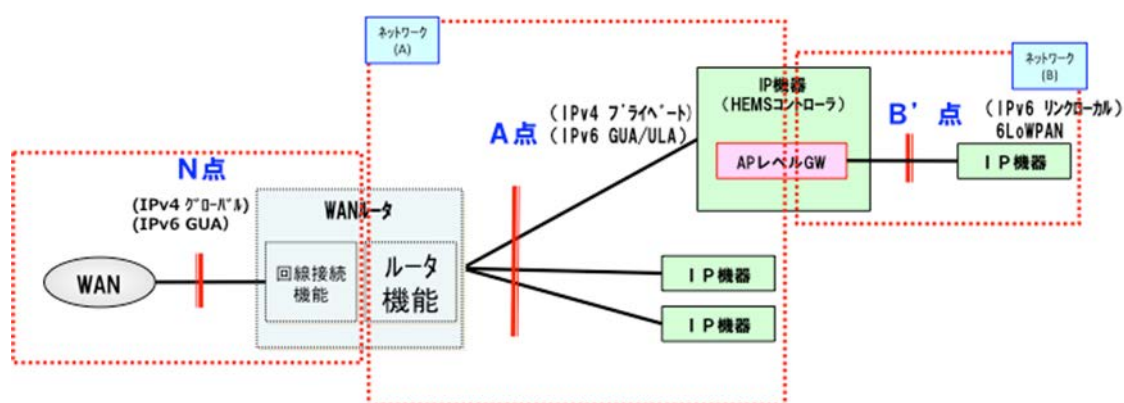


図 1.5 TTC TR-1043 にみるネットワークの構成例 2

図 1.5 に示した 2 つめの例においては、HEMS コントローラは独立したハードウェアとして実装されており、その配下に 6LoWPAN で接続された HEMS 関連の端末機器が接続されている。この例は図 1.3 の定義からすれば素直な実装であるといえ、B'点機器が不用意に A 点接続の IP 機器から見えないような構成を物理的な接続で実現している。

図 1.4 と図 1.5 の 2 つの例は本質的には同じであり、HEMS コントローラがホームゲートウェイ内蔵なのか、独立したハードウェアを持つようになっているのかのみが異なっている。何れも実装形態として現実的なものであり、状況に応じて選択することになる。

以上、この節で見てきたように、図 1.2 における①の接続点は端末機器側のコストなどによる IP への対応の容易性だけでなく、端末機器の性質や利用のしかたに基づいた選択として活用されるものとなる。

1.3 宅内 IP ネットワークインタフェース

図 1.2 における②の接続点は家庭内 IP ネットワークそのものである。現在、ブロードバンド常時接続インターネットは一般家庭にまで普及し、電話回線を用いたダイヤルアップ接続によるインターネット接続はほとんど使われなくなった。家庭内において、各端末機器が互いに IP 接続されているのと同時に、各端末機器は常にインターネットとも通信が可能であるという環境が整っていることになる。これにより、必ずしも家庭内で全ての情報を持っていなくても、必要に応じてインターネット側に取りに行くようなシステムのあり方が可能となった。

しかしながら、家庭における IP ネットワークの構築にはエンタープライズシステムとは状況が異なる点が多々存在する。まず、専用の通信配線を敷設することは新築などの特殊な状況を除けば、多くの場合困難を伴う。その結果以下の様な状況が生じてくる。

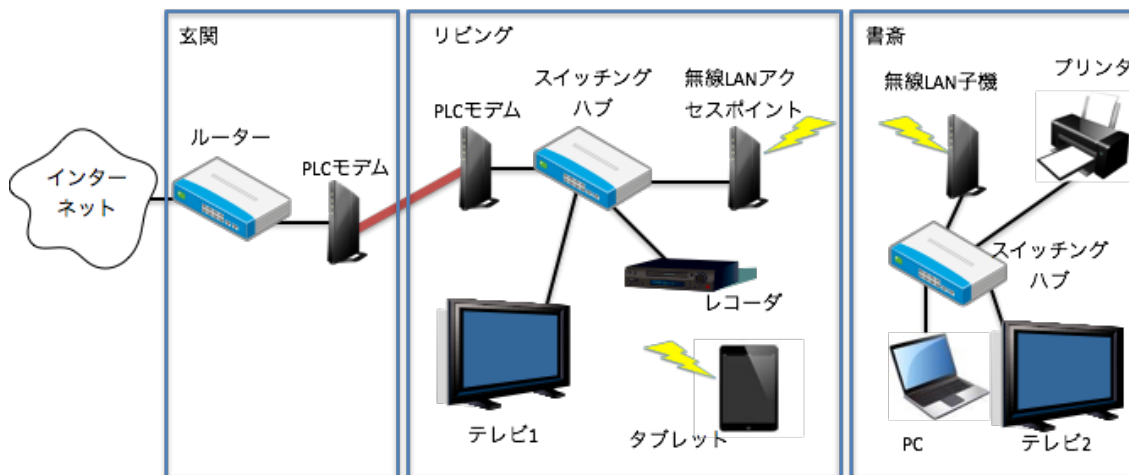


図 1.6 複数の異なるリンク技術を用いて構成されたホームネットワーク例の物理接続

図 1.6 に示すのは、現実の家庭環境を模した接続の例である。インターネット接続の光ファイバの引き込み関連の機器は玄関のブレーカー近くの棚に設置され、そこから **Ethernet** ケーブルを引くことが容易ではないなどの理由で **PLC** を用いてリビングのテレビ周りのラックまでの接続が実現され、そこに **Ethernet** スイッチを置いてテレビやレコーダが接続されるとともに、無線 **LAN** のアクセスポイントも設置している。PC が置いてある書斎へも **Ethernet** ケーブルがないため、無線 **LAN** の子機(Wi-Fi の端末としてアクセスポイントに無線接続し、そこから **Ethernet** へとブリッジする接続機器)を経由して書斎内の **Ethernet** スイッチに接続、その先に PC、プリンタ、テレビが接続されている。

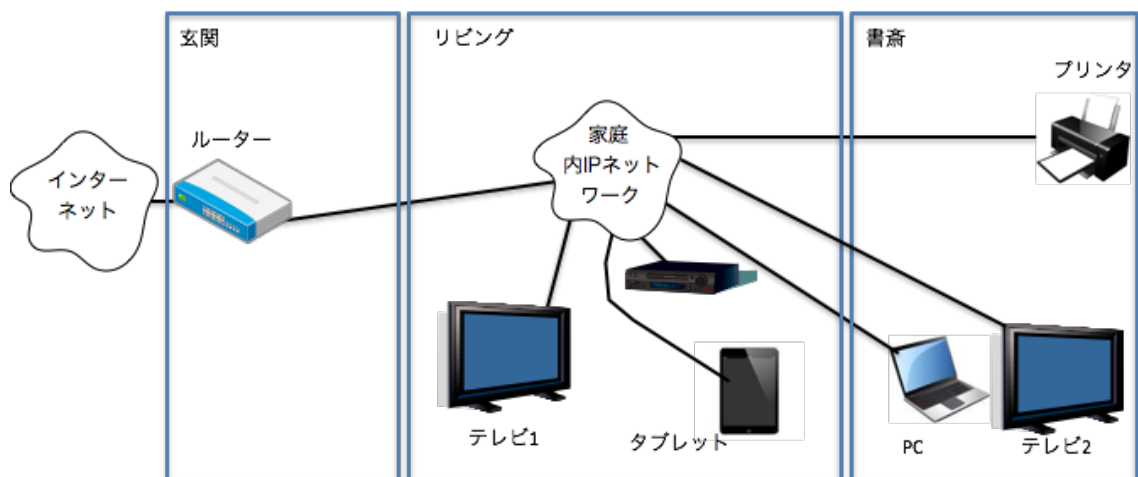


図 1.7 図 1.6 のネットワークレイヤ接続図

この例のように、部屋をまたぐ **Ethernet** ケーブルが確保できないため **PLC** や無線 **LAN** を使うことは極めて一般的に行われているが、何れもがデータリンク層でのブリッジ接続であるため、**IP** レイヤでは図 1.7 のように同一の **Ethernet** ネットワークに各機器が接続されているようにみえることになる。物理的には様々な接続が使われているという事情はわかりにくく、帯域が足りない、ときどき接続が途絶するといったトラブルが生じた際には原因がつかみにくなる。特に注目すべき点は、ルーターの家庭内のインタフェースは有線の **Ethernet** であり、端末機器であるテレビ 1、レコーダ、テレビ 2、**PC**、プリンタ、何れもが有線の **Ethernet** インタフェースで接続されているという点である。通常、ネットワークの問題を発見するのは端末機器であり、端末機器からすれば、自身のインタフェースの種類は判明するものの、相手との間がどのようなネットワークになっているのかについては通常、知ることができない。図 1.6 の場合では、レコーダとテレビ 1、テレビ 2 は何れも **Ethernet** で接続され、図 1.7 を見ても明らかなように、端末機器側からすれば一つの **Ethernet** スイッチに 3 台とも接続されているようにみえる。しかしながら、レコーダとテレビ 1 の間はまさに同一スイッチ下での直接接続であるが、レコーダとテレビ 2 の間は無

線 LAN を間に挟んだ接続となっており、明らかに状況は異なっている。

テレビ 2 のインタフェースが無線 LAN であれば、帯域不足になった際にテレビ 2 自身の無線 LAN インタフェースから接続するのに使われているモード(IEEE802.11b なのか IEEE802.11n なのかなど)や現在の転送レートなどを調べることができるが、図 1.6 のような場合には調べようがなく、単に生じている問題についての警告をユーザーに出すしかなくなってしまう。ユーザーも部屋間接続が無線になっているということを正しく認識していなければ、テレビ 2 が Ethernet インタフェースで接続されていることをもって、レコーダとテレビの間は有線の Ethernet で接続されていると勘違いしてしまう、ということが生じる。これと同様のことは、ルーターとその他の家庭内機器を接続している PLC モデムについてもいえ、故障などで高周波ノイズを電源に出してしまうような家電機器があったときに、インターネットにつながる接続全てに問題が生じるようなことが起きるが、家庭内でのネットワークは正常に動いているため、自分の家ではなくインターネット側の問題であると錯覚してしまう可能性もあるわけである。

このように、家庭内 IP ネットワークはネットワークレイヤとしてみたときには単純ながらも物理レイヤ、データリンクレイヤとしては複雑であり、これらの低位レイヤの管理運用技術は重要なものとなることがわかる。

また、端末機器が単に IP 接続されて通信経路が確保されていれば良いということであれば、ホームネットワーク内に 3G 回線で直接つながる機器が存在してもアドレスさえ通知できれば直接接続されているものと同等にみえることになるが、前節の①のインタフェースの議論でも明らかなように、意図的に接続形態を選択するような場面も多く、また、1.1 でも述べたように、もともとホームネットワークではそれぞれのサービスが独立しているというよりは互いに連携していることが求められ、家庭単位でのネットワークというものが形成される意味がある。

1.4 サービスプラットフォーム-ホームゲートウェイインタフェースとサービスプラットフォーム

フォーム-サービスインタフェース

図 1.2 における③のインタフェースはホームゲートウェイとサービスプラットフォーム提供事業者との間の通信インタフェース、④のインタフェースはサービスプラットフォームとサービス提供事業者の間のインタフェースとなる。これは、明らかにインターネット経由の接続となるわけで、①や②のときのような物理レイヤやデータリンクレイヤに関する議論は意味を持たず、あくまでもネットワークとしては IP 接続が担保されていれば良いということになる。

一方 1.1 でも述べたとおり、サービスプラットフォーム提供事業者とホームゲートウェイ、そしてサービス提供事業者との役割分担という点の方が重要な意味を持つようになる。サービスプラットフォーム型のホームネットワークにおいては家庭内の様々なリソースを一元的に利用可能とし、複数のサービス間での円滑な連携を実現することを目的としている。

その意味では、③のインタフェースは家庭内というローカルネットワーク技術で構成されているシステムを一元的に扱えるようにするためのインタフェースとしての意味を有している。①や②でもみてきたように、各端末機器自体はそれぞれの事情に応じて異なるネットワークインタフェースを有しており、それらの多くは直接インターネットには接続されないだけでなく、物理的な接続距離も家庭内に限られているような技術を用いている。このような端末機器を利用するためにはどうしても家庭内にネットワーク機器を設置し、そこをいわば前線基地として様々な端末機器とプラットフォーム側との橋渡しをする必要が出てくる。この、家庭内におけるネットワーク側の前線基地がホームゲートウェイであり、サービスプラットフォーム提供事業者が最終的なサービスを提供するサービス提供事業者に統一された家庭内へのアクセス手段を提供するものだとなれば、ホームゲートウェイはプラットフォーム提供事業者の前線基地と考えるのが妥当である。

このような考え方に立てば、①や②のインタフェースは、結局ホームゲートウェイが③のインタフェースを提供できるような規格に則っていれば良いということになる。これは、

ホームゲートウェイの仕様を決めるサービスプラットフォーム提供事業者が開発した非 IP のインタフェースのようなものであっても良いが、経済的な効果を考えれば多数のメーカーがつくることのできる標準規格に基づくインタフェースが適切である。この意味で、HEMS 向けの ECHONET Lite や AV 機器向けの HDMI/CEC(Consumer Electronics Control)、DNLA(Digital Living Network Alliance)が①や②で使われる高位レイヤ(アプリケーションレイヤ)プロトコルの代表的なものとなるということは理解できよう。つまり、①や②のインタフェースは機器側の標準を決めるものという位置づけにある。

一方、③は家庭とサービスプラットフォーム提供事業者とを接続するインタフェースであり、いわば家庭から見たホームネットワークサービスの契約インタフェースとなる。ここでは、家庭内の機器に関する情報やコマンドなどがやり取りされるのと同時に、契約的な合意形成のためのやりとりや、サービスプラットフォーム提供事業者からユーザーへの(機器へのではなく)情報の通知といった機能も有する、B to C インタフェースとなる。こうした情報の中でも重要なものに課金関係の情報がある。ユーザーはサービスプラットフォーム提供事業者が提供するサービスメニューから各種のサービス提供事業者のサービスを知り、サービス提供事業者を経由して必要なサービスの契約を行なう。単体のサービスに対する課金は少額であるが、それを合算してサービスプラットフォーム提供事業者は各ユーザーに請求をし、徴収した代金から各サービス提供事業者へと分配を行なう。これにより、サービス提供事業者が直接行なうのは困難な多くのユーザーへの少額課金が現実的なやり方で実現できることになる。

この形態は、以前のフィーチャーフォンや現在のスマートフォンでもみられる。フィーチャーの時代には NTT DoCoMo や KDDI、ソフトバンクといった通信事業者がサービスプラットフォーム提供事業者の役割を果たし、利用者から各種のサービスに対する月額 300 円といった少額の課金を行なっていた。スマートフォンが使われている現在では Apple の App Store や Google の Android Market がその役割を果たしている。Apple も Google もスマートフォンやタブレット内で利用するサービスだけでなくホームネットワークサービ

スに対する興味も示しており、徐々にその体制を整えつつある。

このように考えれば④のインタフェースの位置づけも明確であろう。こちらは、サービス開発者のためのサービスプラットフォームインタフェースであり、アプリを配信するような単にソフトウェアを提供するだけではないサービスの場合には、サービス提供事業者が提供するデータや制御サービスなどをサービスプラットフォームを経由して如何に実現するかというインタフェースになる。こちらは③とは異なり、事業者間の B to B インタフェースであり、エンドユーザーには必ずしも見えないものとなる。

これらの意味で、①や②がサービスプラットフォーム提供事業者と機器メーカー、④がサービスプラットフォーム提供事業者とサービス提供事業者との間の外部インタフェースであり、これらと比べると③は内部インタフェースの位置にあると考えるのが妥当であることが理解できよう。この③のようなインタフェースに立ち入ることなく④や①および②での開発が行えるということは、とりもなおさず、

- ・ サービスのことを考えなくても標準インタフェースに基づいた機器を開発できる
- ・ 機器のことを考えなくてもサービスプラットフォーム提供事業者の API にもとづいてサービスを開発できる

という観点で機器メーカー、サービス提供事業者にとって大きなメリットを与えることになる。これにより、それぞれが独立して開発を進めることができ、サービスと機器の両方を開発する力がなければこの分野に参入できないという状況を大きく改善し、小規模な事業者がそれぞれの得意分野において参入することが可能となるわけである。

1.5 API の抽象度

前節までで述べてきた枠組みにおいて、ホームゲートウェイ、サービスプラットフォーム提供事業者、サービス提供事業者の位置関係は概ね明らかになってきた。しかしながら、実装をどのように行なうかという点においては極めて多くの選択肢があり、その中でも特に考慮せねばならない問題として、API の抽象度がある。①や②のインタフェースにおい

では、ECHONET Lite のようなネットワーク対応家電の機能を扱うためのデータ構造とそのアクセスプロトコルの規格が使われる。つまり、機器付属のリモコンのように、端末機器自体が持つ機能と一致したセマンティクス(意味)でのアクセスができるようなネットワーク・インタフェースである。しかし、ここで使われているセマンティクスをそのまま③や④に適用すべきかどうかについては議論が必要である。

①や②でのインタフェースと同様のセマンティクスで④のサービス提供事業者向け API を構成した場合、サービスを開発するプログラマは、家庭内に設置したコントローラが全ての制御を行なうスタンドアロン型(図 1.1)のプログラムを開発するときと同様に家庭内に存在する端末機器を直接操作するコマンドレベルでのプログラムを書くことになる。これは、端末機器自体が持つ機能のセマンティクスで機器に対して操作することにあたる。例えば、エアコンであれば、モードは自動、冷房、暖房、除湿の何れかで、それぞれのモードにおいて目標温度の値を数値で指定する、という形である。

ここで自動モードにおいては、その意味合いがエアコン本体に依存しており、目標温度もプラスマイナスの気温で指定したり気温自体とは異なる単位で上下をしてするなど、機種によって意味合いが変わる。また、データの与え方が同じであっても基準となる温度が本体 ROM に書き込まれている日付時刻と地域によるものなのか、過去の運転からの学習結果なのかの違いも生じる可能性がある。このような曖昧性が出るようなインタフェースは外部からは使いにくく、明確に冷房、暖房などと指定した運転モードを活用することが想定される。こうして考えてみると、①や②のインタフェースは本来端末機器が持っているインテリジェンスを使わない、より低レベル、つまりは、より具体的な動作に近いレベルのセマンティクスで使われることが予想され、いわば本体内蔵のコントローラをネットワーク接続されたコントローラに置き換え、本体の本質的機構部分(つまりはセンサとアクチュエータ)のみを使おうとする傾向に向かうことが予想される。

このようなプリミティブなセマンティクスで家庭内の機器を使おうとする場合には、④のインタフェースであってもアプリケーションプログラマは家庭内の機器のリストを作成

し、それぞれの機器が持つ機能も DB 化しておき、自分が行ないたい動作をさせるための機器を選んで所定の動作をさせることになる。たとえば、暖房を行なう場合には、暖房という機能を有する機器を選び、その機器に具体的なパラメータ(温度や風量など)を与えて運転し、部屋の温度センサの値を調べながら想定した状況が家庭内で実現されているのかを自らが常に判断するような処理を続ける必要がある。

一方で、④のインタフェースでサービスプラットフォーム提供事業者が「暖房機能」として API を用意し、物理的な暖房機器とは切り離れた機能としての API 提供とすれば、アプリケーションプログラマが行わねばならない処理は、具体的な機器への物理的な操作ではなく、目的とする気温を設定するという論理的な操作のみとなる。各家庭での機器の違いは気にすることはなくなり、本来、アプリケーションで実現したいことに集中した開発が可能となるとともに、同一のコードを機器が異なる家庭に対して利用することも可能となる。この API で呼ばれた機能は、サービスプラットフォーム提供事業者が予め契約顧客の家庭内機器リストとそれぞれの機能の DB を構築して持っていることで、サービスプラットフォーム提供事業者内で具体化され、それぞれの家庭に応じた機器(エアコンだけでなく、オイルヒーターの場合もあるかもしれない)を目標の温度になるように操作するという役割はサービスプラットフォーム提供事業者側がホームゲートウェイも活用して担保することになる。様々なサービスから温度に関するリクエストが来うことを考えれば、④インタフェースでプログラミングする開発者が直接機器を操作するよりも、家庭という単位で契約しているサービスプラットフォーム提供事業者が家庭内環境を整える作業を行なうという方が適切である。また、必要とされる機器リストやその機器に関する DB などは様々なサービス提供事業者がそれぞれ持っているよりもサービスプラットフォーム提供事業者のみが管理しており、API にもとづいて間接的にサービス提供事業者にも使わせているという形の方が、プライバシーの観点でも望ましいものと考えられる。

更に、④インタフェースのセマンティクスを高度化し、暖房して $xx^{\circ}\text{C}$ にするというのではなく、「ユーザーの普段の生活での温度よりも 2°C 高めにする」という API を提供するこ

とも不可能ではない。これを実現するには普段の生活の温度を取得しておく必要があるが、サービスプラットフォーム提供事業者は自社が各家庭に送り込んだホームゲートウェイ配下にある各種のセンサからこうした情報を取得し、自社内(あるいはホームゲートウェイ内)に DB 化しておくことが可能であるため、こうした「普段に対する相対的な」指定を含む API も実現可能となるわけである。同様に、もっと複合的なセンサ活用を考えれば、「ユーザーが汗ばむ程度に」といった指定を行なう API も実現可能となってくる。

こうした複合的なデータでの機能の実現や、過去の記録の活用は一つの家の中だけに限定されているわけではなく、匿名性がきちんと担保できるような利用のしかたをすれば、複数のユーザーのデータから得られた統計値を使ったより高度な機能も実現できる。たとえば、ある特定の地域における平均的な室内温度や、ある年齢層の家族から構成される家庭における室内温度といった観点での指定も可能となる。

このように、④のサービス提供事業向けの API におけるセマンティクスレベルはどれだけの処理をサービスプラットフォーム提供事業者が受け持つかに依存して高度化、抽象化することが可能である。こうした高度な API を活用すればサービス提供事業者の負担は削減され、より多くのサービスが提供されるようになることが期待される。そうなれば、多くのサービスメニューを擁するサービスプラットフォーム提供事業者と契約する家庭が増加することが想定され、より多くのデータが集まり、更に高度な API が提供される、という好循環を招くことになる。逆に言えばこうした正帰還ループに入った先行事業者に追いつくことは極めて難しいものといえ、早急にこうした位置づけの事業者を我が国でも育成する必要がある。

④のインタフェースで機器レベルの操作を直接行わせないようにする方式にはまた別のメリットがある。安全性、効率性の担保である。家全体の環境がどうなっているのかという観点ではなく、特定の家電機器の操作ということで制御を行なうと、単体の家電機器の運転としては正しいかもしれないが、複数のサービスが同時に家庭内で提供されている場合、それぞれのサービスで利用されている家電間の整合性というものは担保されていない。

その結果、人体に悪影響を及ぼすような環境になってしまったり、つながった 2 つの部屋で、片方のエアコンは冷房しながらもう一つの部屋でのエアコンは暖房するようなことが生じうる。サービスプラットフォーム事業者は単に機器の運転の機能を API 化するのではなく、家庭内環境としてどのような状況になるのか、という機能を提供するようにすればこのような問題は解決できることになる。こうした機能をどのように実現するかについてはまだ今後の課題であるが、ホームゲートウェイとサービスプラットフォーム提供事業者のクラウド内サービスとの連携で実現されることは確かであろう。

一方、③のインタフェースにおいて①や②と同様のセマンティクスを使うことは、④での議論とは異なり、ホームゲートウェイがどのような役割を果たすのかという問題に帰着する。もし、ホームゲートウェイにアプリケーションレベルでの機能を持たせず、サービスプラットフォームまでで全ての制御機能を実現していれば、ホームゲートウェイは単にプロトコル変換、つまりシンタックス(形式)変換するだけの存在になるため、セマンティクスとしては③は①や②と同様のインタフェースとなる。この場合には、家庭の中の状況(状態)をサービスプラットフォーム提供事業者かサービス提供事業者側で持つ必要があり、家の中はいわば丸見えの状態にしておくということになる。このような状況を避けるのであればホームゲートウェイにもアプリケーション処理の一部を任せ、③のインタフェースではそれぞれの家庭への方針は与えるものの、どのように実現するのかはホームゲートウェイ配下の機器(例えば HEMS コントローラも含め)で決定するような形になる。

この後者の形態は明らかに家庭内に、ある程度のインテリジェンスを要求することになり、それなりの規模の計算機システムと、そのシステムの運用管理を行なうスタッフが(常駐ではないにしても)求められるようになる。実際、エネルギー管理システムの分野では、大きなビルを対象とした BEMS(Building Energy Management System)では、外部から緊急の節電要請などが BEMS のコントローラに通知され、BEMS コントローラではそのビル内でどの資源をどう運転して要請を満たすか、様々な選択肢がある中から最適なものを選んで実行する。更に、その後も想定よりも電力が下がらないなどの状況が起き

ないか監視しながら、場合によっては更なる手を打つ、という役割も BEMS コントローラが果たす。しかし、家庭向けの HEMS の場合にはこうした抽象度の高い依頼が外部から来て内部で具体化するという手順をとるというよりは、逆潮流をやめるであるとか、その期間中は蓄電池電力を使うなど、より具体的な対策が外部から HEMS に送られることが想定されている。

こうした違いは、③のインタフェースでどれくらい標準化した情報伝達が可能であるかということにも依存する。BEMS ではこのようなインタフェースにおいて利用すべき情報モデルとして ASHRAE FSGIM(American Society of Heating,Refrigerating,and Air-Conditioning Engineers/Facility Smart Grid Information Model)や IEC61970 CIM(Common Information Model)など、様々なドメインで使われているデータ構造に基づいた相互連携のためのしくみを多大な労力をかけて開発し、実現しようとしている。同様の努力を HEMS に対して行なうのかは、今後、我が国でも OpenADR によるネガワット取引がなされるのに向けてどのように議論が進むかに大きく依存する。また、これらの話はエネルギーマネジメントという特定のサービスに特化した話であり、見守り、セキュリティ、エデュテイメントなど、様々な分野でこのような高次概念の整理が行われ、それに基づいて標準化されたプロトコルが実装されて相互接続性が実現されるにはかなりの期間を要することになる。これが、1.1 の冒頭で③のインタフェースはしばらくはサービスプラットフォーム提供事業者の内部インタフェースとなろうと書いた根拠となっている。

1.6 サービスプラットフォーム型に向けての道筋

1.5 で述べてきたように、サービスプラットフォーム提供事業者の機能を高度化することで、統計値を使った新しい機能が提供されたり、機器構成や家族の年齢が異なる家庭にも同一の API でプログラミングできたりするなど、様々なメリットが生まれる。しかしながら、こうした環境を実現するには技術的な開発、ユーザー側の認知度の向上、各種のビジネスモデルの開発、法令の改正など、様々な困難な問題が立ちはだかる。

そのような状況下においても可能な範囲でホームネットワークサービスを実現していくためには、サービスプラットフォームというものを念頭には置きながらも現実を見据えた開発が必要となる。その中には、ASP 型のようにみえる形態において、一つのサービス提供事業者が複数の分野のサービスを提供し、内部的に共通できる部分を徐々にプラットフォームとして切り分けていく、という形もありうるかもしれない。また、逆に最初からサービスプラットフォーム型の形態はとるものの、アプリケーションに関連したサービスはサービスプラットフォーム提供事業者は実現せず、プラットフォームの上にプラットフォームを実現するようなモジュール事業者がサービス分野ごとに出現してくるという形もありうるものと思われる。何れにしても、この形態を実現してデータ蓄積をいち早く開始できるようにすることは重要である。

このような、様々なステークホルダを集めた議論を行ない、標準化につなげる場として、スマート IoT 推進フォーラム[2]が 2015 年末に設置されており、2016 年度より活動を開始している。

第2章 ホームネットワークのセキュリティ

ホームネットワークを含めた IoT システムには、2 種類のセキュリティがある。一つはサイバーセキュリティとしてのセキュリティであり、もう一つは物理的な動作を伴うことに関するセキュリティである。IoT セキュリティについてはその概念自体がまだ新しく、今後その概念の整理も含めた議論が深まって行く段階にある[3-7]。本章では、こうした IoT セキュリティをめぐる現状について述べる。

2.1 情報セキュリティからサイバーセキュリティへ

情報セキュリティに関する議論は古くから行われており、その成果は明文化されて国際標準規格にも定められている。ISO/IEC 27000 シリーズの国際標準(国内では JIS Q 27000 シリーズ)では、情報セキュリティを確保するとは以下の 3 点を満たすことと定められている。

Confidentiality (機密性)

情報へのアクセスを認められた者だけが、その情報にアクセスできる状態を確保するような特性

Integrity (完全性)

情報が破壊、改ざん又は消去されていない状態を確保するような特性

Availability (可用性)

情報へのアクセスを認められた者が、必要時に中断することなく、情報及び関連資産にアクセスできる状態を確保するような特性

Confidentiality を満たすための技術例にはアクセス制御、暗号化などがあり、Integrity

を保つためにはデジタル署名が使われる。また、可用性を上げるためには資源の多重化をはじめとした各種のディペンダブルシステム技術が用いられる。

これらの頭文字をとって、「セキュリティの CIA」という表現もよく使われる。さらに同規格では、以下の 4 点についても含めて定義しても良いとしている。

authenticity (真正性)

利用者、プロセス、システム、情報などのエンティティが、想定した通りの振る舞いをすることを確実にする特性

accountability (責任追跡性)

あるエンティティの動作内容を一意に追跡できることを確実にする特性

non-repudiation (否認防止)

ある活動又は事象が起きたことを、後になって否認されないように証明できるようにする特性

reliability (信頼性)

意図した動作と結果が一貫していることを確実にする特性

CIA の三要素を必須とし、これら 7 要素を備えたシステムがセキュアなシステムということになり、それぞれの要素について定性的あるいは定量的な尺度で評価を行いながらその実現を目指す形になる。しかしながら、それにどれくらいのコストがかかるのかは、そのシステムをとりまく状況の変化に応じて変わることになる。

例えば、confidentiality の場合、コンピュータシステムの素人(あるいは意図的に悪いこ

とをしない利用者)が相手であれば簡単なパスワードだけで実現が可能かもしれないが、高度なシステムの知識を使って本来認められない情報へのアクセスを試みる攻撃者がいれば、強固な暗号化を行なう必要が生じてくる。この暗号化も、解析技術や解析に使われるコンピュータシステムの能力が向上すれば、それに応じて更に強固なものを用意する必要が出てくる。

また、パスワードを正面から破りに行くような技術的な攻撃ではなく、そもそもパスワードは人間が決め、人間が入力する事が前提であることに着目して、人間からパスワードを聞き出そうとするような攻撃を行えば、いかに強固な暗号で保護されたシステムでも容易に破られてしまうことになる。こうした攻撃はソーシャル・エンジニアリングやソーシャル・ワークと呼ばれ、現在では日常化している。

同様に、availability の場合には、機器の故障やソフトウェアのバグなどでシステムが止まることを想定した場合には機器の二重化で対処が可能となるが、意図的に処理能力を超えるような大量のアクセスを行なってサービスをダウンさせるような攻撃がなされるようになれば、二重化では解決にはならなくなる。

意図的なセキュリティの攻撃は、当初は自己の技術力を顕示したい個人による愉快犯的なものが多かったが、ネットワーク接続された計算機システムが社会のインフラを担うようになると、意図的、計画的に、大規模な組織が攻撃を行なうようになってきた。特に、国家間での攻撃は既に戦争の様相を呈しており、国によっては国内の最も優秀な人材をそれに当てるようになっているところもある。

こうした状況のもとでは従来の情報セキュリティの枠組みを超えた対処が必要となり、その呼び方もサイバーセキュリティという表現を使うようになってきた。これは、サイバー空間における活動が我々の日常生活や経済活動にとって必須のものとなった現在、サイバー空間内における治安維持、防衛活動が必要となっている、ということを意味しており、我が国においても 2013 年以降、内閣府の組織が情報セキュリティからサイバーセキュリティを対象としたものに強化されている。

セキュリティに対する攻防が高度化するほど、その対応に関する考え方にも変化が生じている。たとえば、パソコンやサーバーから構成される企業等のエンタープライズシステムにおいても、以前はファイアウォールで危険の存在する外部ネットワークと安全な内部ネットワークを明示的に切り分け、各パソコン単位においてもアンチウイルスソフトやパーソナルファイアウォールで問題の原因となるものをシャットアウトしようとする考え方が中心であった。しかし、攻撃側の組織が大規模化し、極めて高い技術レベルを有し、ソーシャル・エンジニアリングをも駆使した手口で攻撃してくるとエンタープライズシステム内にマルウェアに感染したり、乗っ取られたりしているコンピュータが全くない状態に保つことは現実的ではないほど難しくなっている。

この状況を受けて、従来行われてきたシステム内に汚染されたコンピュータがないようすべく、入口側でシャットアウトしようとするのではなく、もうシステム内に汚染されたコンピュータがあることを前提に、それでも機密情報が流失したり、重要なデータが破壊されたりしないようにしようという、出口側での対処を考えるように考え方も変化している。こうした対処のことを出口対策と呼んでいる。

近年猛威をふるっている持続的な標的型攻撃(Advanced Persistent Threat: APT 攻撃)では、主にソーシャル・エンジニアリングや多くの人が日常的に参照している Web サイトの未知な脆弱性を使ってエンタープライズシステムの内部に悪用できるコンピュータを確保し、数カ月から数年に及ぶ潜伏期間の間に社内での情報の流れや定型的に使われるメールの文面を調べる。そうして得られた情報を活用して、本物と見分けがつかないような精巧な偽造メールを作成し、ターゲットとするユーザーに送付するが、攻撃されるユーザーにとっては、社内や取引先の信用している人物からの連絡メールを受け取ってその指示に従ったり添付文書を開いたりするだけでマルウェアに感染させられ、機密文書が漏洩したり、サーバー内のデータが破壊されたり暗号化されて読み出し不可能になったりする、という状況に追い込まれることになる。

このように、インターネットを中心に発展してきた情報システムから構成されるサイバ

一空間においては、既に高度な犯罪的行為が日常的に行われており、その対処に多大なコストがかけられているのが現状である。ホームネットワークも、かつては家庭内での接続のみに閉じたネットワークであったが、クラウドや他のサービスとの連携を前提とするようになり、サイバー空間と連続した存在になりつつあり、サイバーセキュリティの脅威がホームネットワークにも及ぶことは避けられない状況にある。

2.2 IoT セキュリティ

昨年度の報告書第2章でも述べたように、ホームネットワークを含めたIoTシステムはそれまでの情報システムとはいくつか異なった点がある。従って、セキュリティに対する考え方にも新たな観点が加わって然るべきである。

まず、ホームネットワークのようなIoTシステムの特徴についてまとめてみよう。昨年度報告書の第1章で紹介したアーキテクチャおよび同第2章で紹介したシステムを構成する5つの要素技術についての記述からも明らかなように、ホームネットワークには以下のような特徴が存在する。

- | | |
|-----|------------------------------------|
| 特徴1 | システム内でネットワーク接続されるノードが極めて多い |
| 特徴2 | ネットワーク接続されるノードの計算機としてみたときのリソースが乏しい |
| 特徴3 | 互いに別々に開発された多数のシステムが相互接続して全体を構成している |
| 特徴4 | 実空間との物理的なやりとり(センシングとアクチュエーション)がある |
| 特徴5 | サービスそのものが、ユーザーの生活と密着している |

これらの多くはホームネットワークに限らずIoTシステム全般に共通していることであり、これらのうち、ホームネットワークに特有なものは特徴5のみであるといえる。以下、

これらについて概観する。

特徴1 システム内でネットワーク接続されるノードが極めて多い

IoT には、既存のモノを情報ネットワークで相互接続するというイメージがあるが、一方で、既存のモノは単体で行なっていたサービスを複数のネットワークで接続されたノードで実現する、という意味を持っていることには注意が必要である。たとえば、現在の高級エアコンはエアコン単体の中に部屋中の温度をセンシングする機能が入っており、単独で最適な空調制御を行おうとする。一方、ホームネットワークでの実装においては、部屋の各所に取り付けられた温度センサがエアコンにその情報を与えるため、エアコン自体は高度な温度センサを持つ必要がなくなる。また、この温度センサはエアコンだけでなく、健康管理や異常検出など様々なサービスで共有することが可能である。

このように IoT 化、ホームネットワーク化することにより、エアコン単体の価格は下がることになるが、高級エアコンではセンサ機能も含めて一つのネットワークノードという扱いであったものが、センサの数だけネットワークノードが増えることになる。現在の高級エアコンでは部屋の中を数千区画に分けて温度計測をしており、同様なことをしようとするとネットワークセンサの数も相当数になることを考えればネットワークに接続されるノードの数は極めて大きなものになることは容易に想定できる。

他の例では、現在スマートメーターの設置が我が国でも進められているが、メーターが取得した情報を家庭内の HEMS コントローラでも利用できるようなインタフェース(B ルートインタフェース)が設けられることになっている。国内のスマートメーターの個数は低圧需要家だけで最終的に 7,750 万個におよぶと見込まれており、全ての電力事業者において B ルートインタフェースとしては無線(Wi-SUN(IP))および PLC(G3-PLC)が採用されているため、いわば、日本中に 7,750 万個の無線インタフェースが同一規格で口を開けることになる。無線や PLC のような外部への漏洩の可能性のある技術を用いたインタフェースがこれだけの数、外部に晒されることは今だかつてない状況といえる。

ネットワークに接続されるノードが増えるということは、セキュリティ対策を施さねばならないノードの数が増えるということ以外に、乗っ取られた際の影響が大きくなるということも意味している。センサノードのようなものを考えても明らかなように、同一規格のネットワークノードが極めて多数利用されるような形態が想定されるため、一度脆弱性が見つかりと多数のノードに対しての攻撃が可能となる。侵入した攻撃者が以後攻撃用途に利用できるノードとして利用できるような乗っ取りができてしまうと、攻撃者側は膨大な攻撃ツールを手にするようになる。乗っ取りまで行かないにしても、増幅型 DDOS 攻撃と呼ばれる手口で利用できるような挙動を示すようなノードであれば、それだけで攻撃側にとっては極めて有効な資源となってしまう。特徴 2 で指摘するように、IoT システム内の各ノードは計算機としてみた場合の余力がなく、十分なセキュリティ対策ができない可能性があることを鑑みれば、この点は重要である。

特徴 2 ネットワーク接続されるノードの計算機としてみたときのリソースが乏しい

前項目でも述べたように、IoT システムにおけるネットワークノードには、センサノードのように今までは本体の一部品として組み込まれていたようなものがネットワークとして接続されるようになったものが存在している。これらのノードはそれが果たすべき「本体機能」を鑑みると計算機としての能力は必要なく、最低限ネットワークに接続できるようになるための能力があれば良いようなものがほとんどである。

このようなノードにおいては、ファイアウォールのような機能を自前で持つことはおろか、自らのソフトウェアをアップデートできるような機能もコスト的な制約から難しいような場合も多い。システムのセキュリティ強度は最も弱いところで規定されるという性質から考えても明らかなように、こうしたノードはシステム全体におけるウイークポイントとなる可能性が高い。

とはいえ、全てのノードに十分な計算機リソースを与えることはコストの面で無理があり、他の手段で問題を解消する必要がある。もっとも有望な方法はゲートウェイデバイス

を利用することである。ゲートウェイデバイスは十分な計算機リソースを有し、自らのソフトウェアアップデートや挙動の監視などを行なう能力を有するとともに、配下にある脆弱なネットワークノードを外部から隠蔽する役割を果たす。このようなゲートウェイデバイスを用意すれば、センサノードのような小規模ノードではインターネットプロトコルとは異なるローカルのネットワークプロトコルを実装することでコストを下げることも可能となる。この場合、ローカルのプロトコルを使うノードにはインターネット側から直接通信する手段もなくなるという効果もある。

また、ゲートウェイデバイスは単に接続上の橋渡しをするのみならず、ゲートウェイ上にインテリジェントな機能を持ってエンドノードからの通信量を削減することも可能である。たとえば、配下の全センサの情報を上位に上げるのではなく、ある条件にマッチしたときのみ通信する、あるいは、一旦ゲートウェイ上に記録し、要求があったデータのみを上げるなどの利用のしかたが考えられる。もちろん、ゲートウェイ自身が高度な制御機能を持って、センサやアクチュエータを利用し、おおまかな運転方針のみを上位のノードとやりとりするような使い方も可能である。一方で、こうした実装形態ではゲートウェイデバイスの信頼性は極めて重要であり、ゲートウェイデバイスに問題が生じると配下のノードの機能全てを含めた障害が生じてしまうことになる。

特徴 3 互いに別々に開発された多数のシステムが相互接続して全体を構成している

セキュリティの対策を考える際には、対象となるシステムの仕様が明確であることが重要である。各部における挙動が明確に定義されていれば、それからの逸脱をもって異常状態を検出することも可能となるし、脆弱性を検討する上でシステム内各部の技術要素が明示されていればそれに対応した技術的な対策を行なうことができる。



(a)直接ケーブルで接続されている場合



(b) 間に様々な接続技術が使われている場合

図 2.1 機器間にネットワークという「システム」が入る

しかしながら、IoT システムにおいてはシステムを構成する各部品自体が既に複雑なシステムとなっていることがほとんどである。ホームネットワークシステムの場合であれば、構成要素も比較的単純なものが多いとはいえ、たとえばネットワーク接続が図 2.1(a)に示すように直接 Ethernet ケーブルで接続されているのか、図 2.1(b)のように間に様々な通信技術が挟まっているのかは接続されているエンドノードにはわからない。図 2.1(b)の場合には、左から、一旦 Ethernet で出たケーブルが PLC のようなモデムに接続され、更にモデムからインターネットを超える VPN ルーターを経由して他のネットワークへと接続され、そこから更に無線 LAN を経由して再び Ethernet に戻って接続されている。この場合でも、L2 接続で同一サブネット(ブロードキャストドメイン)内に存在するような設定は可能であ

り、機器から見た場合には図 2.1(a)と同様の動作となる。しかしながら、間には媒体共有型の通信を行なう PLC と無線 LAN が入るうえ、インターネット内も経由しているという点で、セキュリティ上は全く扱いを変えねばならない。

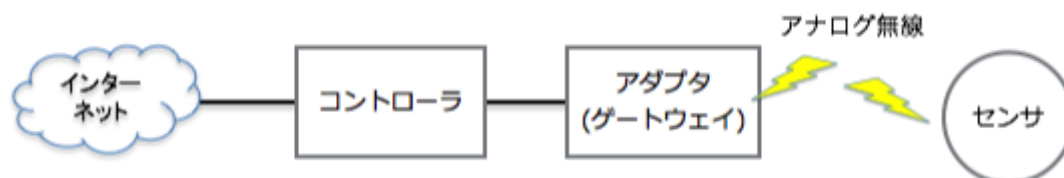


図 2.2 センサがアダプタでネットワーク化されている例

図 2.2 に示したのは、アダプタ(ゲートウェイ)経由でセンサが接続されている例である。この場合、コントローラから見るとネットワーク対応のセンサとしてみえ、インターネット接続部分やコントローラとネットワークセンサ間の接続をいくら強固なセキュリティを有するものにしても、末端の部分でアナログの無線が使われていれば、ここで傍受が可能となってしまう。

何れの場合においても、ネットワークノードは状況を把握することが不可能であるという点が重要である。このようなシステムの場合、全体のセキュリティ強度を保つことはできず、結局のところ、それぞれの構成要素について実装方法、あるいはセキュリティ強度を規定するような形で対処するしかない。たとえば、図 2.2 の場合には、末端のセンサとコントローラの間でのセキュリティ強度を定め、その性質を満たすようにアダプタとセンサとの間の実装方式を決めるようにするわけである。とはいえ、全ての場合を想定した設計は困難であり、設計の方法論等、今後の大きな課題の一つとなっている。

特徴 4 実空間との物理的なやりとり(センシングとアクチュエーション)がある

これは、IoT システムの本質的な特徴であるといえる。ホームネットワークも家電のように、明るさや気温など、物理的な空間の情報をセンシングしたり、それを変更するというアクチュエーションをしたりするもののネットワークであり、一般に IoT システムでは

様々な物理量を計測したり変更したりすることになる。

これはそれぞれ、

センシング： ユーザーが意図的にシステムに入力したわけではない情報が取得されている

アクチュエーション： システムの稼働結果が直接物理的な作用として現れる
ということを意味している。

センシングにおいては、従来のコンピュータシステムは、キーボードやタッチパネルなどで意識的に入力した情報にもとづいて稼働していたため、サイバー空間の中にどのような情報が入力されるのかはある程度把握することができた。

しかしながら、ホームネットワークでは、人感センサやドア開閉センサ、また、電力消費の状況により、いつトイレに行き、トイレ内に何分間滞在したかなど、今までは意識していなかったような情報も取得が可能である。また、これらの情報は、システムが稼働していることを認識していない家族の分も取得できているという点も重要である。

これらの情報は健康管理や省エネなどには必要なものではあるが、情報がシステム側に取得されているという意識のないユーザーの就寝時も含め 24 時間 365 日の情報が取得されているという状況は、今までとは様相が異なるということは明らかであろう。

アクチュエーションについても、状況は従来のコンピュータシステムとは大きく異なることになる。サイバー空間の中での出力は情報という形であって物理的な作用そのものではないため、直ちに直接人体に危害を加えるようなことは生じにくかったが、IoT システムにおいては物理的な作用を及ぼすアクチュエータが存在するため、直接人体に危害を加えることが起こりうる。これは自動運転やロボットアームのような大規模なものに限った話ではなく、ホームネットワークにおいても、エアコンの運転を悪意を持って操作することにより、熱中症のような状況を生じさせ、ユーザーを死に至らしめる可能性がある。我が国では毎年数十人の高齢者が自宅で夜中に熱中症でなくなっており、こうした状況を人為的につくって殺害を図ることは十分に実現可能なものである。

こうした、センサやアクチュエータが直結したシステムならではの脅威に対してどのような対策を講じるべきかという点については現状はまだ議論が深まっているとはいえない。前述のようにエンタープライズシステムのセキュリティにおいても出口対策という概念が用いられており、IoT システムにおいてもサイバー空間内の要素に対する侵入が行われていても、結果としてシステム全体では人体に危害を加えないようにするためには、サイバー空間内での対策でアクチュエータに対して危険なコマンドが行かないようにすると同時に、アクチュエータ自身についても人体に直接危害を加えるような動作はしないようなしくみが必要とされるであろう。これについては、2.3 で再度とりあげる。

特徴 5 サービスそのものが、ユーザーの生活と密着している

ホームネットワークにおけるサービスの利用はユーザーの生活そのものである。従って、特徴 4 でも述べたように、ホームネットワークシステムはユーザーの情報を就寝中であっても取得し続けているし、ホームネットワークの動作の不調は生活そのものに直接影響を与え、場合によっては危害を与えるに至ることもある。

この状況は、他の IoT システムと比較しても厳しいものであると考えられる。全ての労働者は家庭における生活者であり、家庭環境という足元からの攻撃は全ての産業に影響を与えてしまうということもできる。また、新生児から在宅介護中の高齢者まで、全ての人類がホームネットワークのユーザーとなる可能性があり、安心・安全が保たれなければその存在意義自体に疑念が生じてしまうようなものと考えられる。

2.3 今後に向けて

以上のように、セキュリティの観点でホームネットワークシステムには既存のセキュリティ検討対象となっていたシステムと比較して異なった点があるが、ここまでの議論でも明らかなように、これらの中には本質的に意味合いの異なる 2 つの課題が存在している。

特徴 1 から 3 までは、サイバー空間の中の議論であり、既存のサイバーセキュリティの

議論の枠組みで対処が可能である。一方、特徴 4 と 5 は実世界とサイバー世界が密接に絡みあう IoT システムの性質と、対象が家庭環境であるというホームネットワークの性質から来るものである。

特徴 1 と 2 は本質的にはコストの問題であり、費用対効果を考慮して落としどころをつけることになる。今後、IoT システムが一般化し、様々な分野で使われるようになれば、出荷されるチップやモジュールの数量は従来型のコンピュータシステムとは比較にならないほどの数となり、価格の低減が期待できることは明らかである。現在は家電を含め広く使われている Ethernet インタフェースも 1980 年代には数十万円の特設なインタフェースカードを装着するようなものであったことを考えれば、普及の流れがついたことで価格が劇的に低減した事例も既に存在している。

特徴 3 は IoT システムに限らない、近年のオープンシステムの抱える問題であり、様々なアプローチでの解決が図られていくものと考えられる。

特徴 4 に対する考え方は前述のようにまだ定まっていない状況にある。2.2 でも述べたように、アクチュエータを有するシステムにおける出口対策というものはどのようなものであるか、という議論は重要であるが、これと極めて類似した議論は過去にも行われてきている。その一つが、組込みシステムにおける機能安全である。組込みシステムはホームネットワークシステムとも極めて密着な関係を有しており、ホームネットワークシステムの各要素、特にネットワーク家電は組込みシステムの代表的なものであり、ホームゲートウェイも組込みの枠組みで議論されている。家電はそもそもが電気機械として冷蔵や洗濯のような本体機能を実現する電化製品が母体としてあり、その運転や機能を高度化するためにマイコンが内蔵されて組込み機器となった。いま、そこにネットワークとの接続機能が追加されてネットワーク家電となり、ホームネットワークの一要素となるに至っているわけである。

一般に危険が及ぶような状況をなくしてしまうという考え方を「本質安全」という。よく引き合いに出されるのが、鉄道の踏切で、これを立体交差にしてしまつて踏切自体をな

くせば踏切事故はなくなる。家電においても、発熱する家電について、発熱温度自体が危険な温度にならないようにしてしまえば、本質安全を達成することになるが、これは、電気ストーブでセラミックヒーターを使うと高温にならずに一定温度の温熱を生成できる、というような例はあるものの、通常は本体機能が満たせなくなってしまう、達成は難しい。一方で、安全を担保するための機能を追加するという考え方が「機能安全」である。温度ヒューズや電力ヒューズのようなものも、一定の条件になれば動作を停止するという制御の一種である。現在の機器においては、マイコンを利用して、より高度な制御を行なうことが可能であるため、マイコンなしでは実現できなかったような運転も可能となっている。一方、現在のマイコン内蔵の家電は安全のためだけにマイコンを内蔵したのではなく、そもそも追加したい機能があったためにマイコンの内蔵に至っているのが普通である。こうした追加機能が新たに危険性をもたらす可能性がある。このような、追加機能に対する安全を担保することも、機能安全と呼ばれる。機能安全に対する研究開発は、組込みソフトウェアの重要性が高まるのと平行して進められており、我が国でも IPA(独立行政法人 情報処理推進機構)がガイドラインをまとめて公開している。この機能安全とセキュリティとの位置関係についての検討が近年関心を集めるようになってきている。欧州では SESAMO(Security and Safety Modeling)プロジェクトが 2012 年から 2015 年にかけて行われ、我が国でも IPA や国立研究開発法人産業技術総合研究所が取り組みを行なっている。

一方、安全という分野と関連は深いものの異なる分野に高信頼という分野がある。重要な社会インフラを支えるコンピュータシステムはほぼ例外なく二重化以上の多重化がなされたシステムであり、ハードウェアの故障による停止は起こらないようになっている。メインフレームと呼ばれるコンピュータでは、プロセッサ単体レベルでこのような機能を有しており、故障による障害を抑えるような仕組みが組み込まれている。

この信頼性と情報セキュリティとの間の関連については古くから意識されており、図のように、用語の整理もなされたりしているものの、ディペンダブル(高信頼)コンピューティングの世界と情報セキュリティの世界との間でのコンセプトが必ずしも一致しているとは

言い難い状況にある。

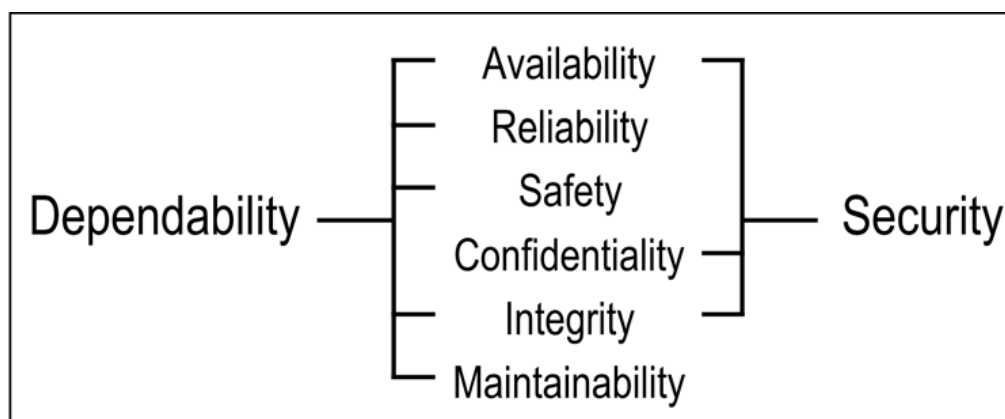


図 2.3 ディペンダブルコンピューティングと情報セキュリティでの属性を示す用語

このように、セキュリティ、安全、高信頼は互いに深い関係は持ちながらもそれぞれが発展してきたバックグラウンドの違いなどにより、用語や概念の整合性から、相互の関係の考え方まで、検討すべき点が多々残っている。特に、情報セキュリティからサイバーセキュリティへと状況が変わったり、組込みシステムとインターネットという従来はあまり接点のなかった分野が融合して IoT システムになったりと、状況が大きく変化していることを受けて、様々な概念の見直しを行なう必要もあるかもしれない。

特徴 5 は、ホームネットワークでは避けられない問題となるが、ホームネットワーク以外の情報通信システムでも問題になりつつある分野が出てきた。現在もっとも顕著なのは、スマートフォンとスマートウォッチである。従来型の携帯電話とは異なり、スマートフォンは小型のパソコンとなっており、アプリケーションの適用範囲が極めて広いため起床時から就寝に至るまで、常に利用するようなユーザーも珍しくない。このような状況ではスマートフォンから取得できる情報は個人の活動を極めて詳細に描き出すものとなる可能性が高い。これを積極的に活用しようとする「ライフログ」に関する検討も行われている一方、パーソナルデータとしてプライバシー保護をどのように考えるかという議論も巻き起こっている。スマートウォッチは、現在のところはスマートフォンの周辺機器のような位

置づけにあり、これを利用することで、より密着してユーザーのデータが取得できるようになる。たまたまスマートフォンを手放している状況においても、スマートウォッチは身に着けている状態にあるし、スマートフォンでは取得が容易ではなかったユーザーの心拍数や活動状況も取得が可能となる。

こうした新しいデバイスで収集される情報は、ユーザーにとってメリットのあるサービスにつながることは確かであるが、悪用されるリスクとメリットを明らかにし、コストをどれだけかけてリスクを低減したりメリットを増加させるか、といった落としどころを検討していくことになる。この背後には、システムが取得した自分の情報は一体誰が所有すべきものなのか、という問いが潜んでいる。米国では、例えば Amazon 社に対し、ユーザーの購買行動履歴情報はユーザー自身に所有の権利があり、この情報を用いて Amazon 社が利益を上げるのであれば、ユーザーに利益の一部を還元すべきであるという訴訟が行われている。一般に、ユーザー自身の情報は、企業等、誰かがシステムを構築して収集・蓄積しなければ元々すぐに消えてしまったものであると考えられ、この点ではコストをかけて収集した企業が所有するのが妥当なように考えられる。しかし、一旦蓄積されたデータからはユーザー自身の行動が浮かび上がり、プライバシーの問題も生じる性質のものであることも事実で、やはり情報はユーザー自身の所有とする方が妥当なようにも考えられる。一方、見方を変えれば本来は消えてしまった情報を企業が価値のあるものとし、企業からその利益の一部が直接還元されるか否かに関わらず、ユーザーに対する新しいサービスが実現され、これがユーザーに大きな利益をもたらすのであれば、ユーザー、企業、ともにメリットのある状態と考えることも可能である。既に定着しているカーナビゲーションのテレマティクス情報の提供などは、この一例とみなすことができる。こうした「パーソナルデータ」に関する議論はインターネットの一般利用が盛んになった今世紀初頭から続いており、純粹に技術的な課題というよりは、社会やユーザーの受容性のような性質のものであり、様々な事案を積み重ねながらコンセンサスが醸成されていくものであると考えられる。法律のようなあたかも決まった規則のように見えるものでも、法解釈と

いうものはその時代背景に応じて変化している。現在では酒酔い運転は厳罰の対象であるが、50 年前は酔って運転し水田に突っ込んだという話は笑い話として扱われていた。データの利用のしかたについての社会的なコンセンサスも大きく変化していくものと考えられる。制度的にも様々な試行が行われている。英国のマイデータ・プログラム、米国のスマート・ディスクロージャを始めとした政策の動きもあり、今後も動向に注視する必要がある。

この議論でもう一点注意しておかねばならないのが、セキュリティとプライバシーとの違いである。パーソナルデータのようなプライバシーを含んだ情報の取り扱いにおいて、Confidentiality が保たれず、外部に流出してしまうような問題が生じればこれはセキュリティの問題である。しかしながら、プライバシーに関わるパーソナルデータを取得すること自体、また、それをどのように(何の目的で)使うのかということも問題を引き起こす。我が国でも 2013 年に起きた交通系 IC カードの履歴をめぐる問題では、システムに脆弱性があったて乗車履歴が外部に流失した結果、特定の個人がどのような経路で通勤しているのかが明らかになってしまった、などという問題ではなく、IC カードのユーザーと企業側の契約条項にもとづき、企業側が取得したデータから新たなサービスを実現しようとしたところ、そのような「目的外」の使い方はユーザーの意思に反する、という反発を受けたものである。つまり、セキュリティ上の問題がなくてもプライバシー、パーソナルデータの問題は生じるわけであり、セキュリティと比較してもプライバシーの問題は社会的、ユーザー側の心理的な要素が強いものと考えられる。

本章では、ホームネットワークのセキュリティに関して述べてきた。ホームネットワークに限らず IoT システムのセキュリティは明らかにしていかなければならないことが多数残されている状況であるが、現実の社会で活用されていくためには乗り越えていかなければならない課題である。2.2 で述べたように、ホームネットワークの分野では、大量のスマートメーターが導入されつつあったり、2016 年度からは太陽光発電のパワーコンディショナー(DC/AC 変換をしたり、発電した電力を配電線に送るのか家の中で使うのかを制御する装

置)にネットワークインタフェースが取り付けられ、制御可能な範囲を超える逆潮流を家庭の側で抑制できるようなしくみが導入されるなど、着実にアクチュエーションを伴うサービスが浸透しつつある。これらに対する攻撃は停電という形で顕在化し、社会に大きな影響を与えることとなる。

今までホームネットワークに対するサイバー攻撃が何か事故を起こしたという報道は聞かれていないが、これにはいくつかの理由があるものと考えられる。ひとつには、ホームネットワークシステム自体が発展途上のものであるため、各システムの技術がバラバラであることが考えられる。それぞれ別の脆弱性をつかねばならないとなると、攻撃側からすれば開発の効率は悪く、対象として取り上げにくいということになる。しかしながら、ホームネットワークシステムにおいても、末端の機器インタフェースのようなところにもインターネットプロトコルを導入したり、クラウド側のサービスとの接続部分などに関してミドルウェアの標準化が進展していることを鑑みれば状況は変化しつつあるといえる。

もうひとつの理由には、攻撃する価値が高くなかったという点が挙げられる。組織だった攻撃の対象となるシステムは、攻撃により得られる情報が何らかの経済的価値を持つものであったり、社会のインフラシステムなど、テロ攻撃の対象となるようなものが選ばれていた。この点ではホームネットワークシステムは今までは該当しなかったものと考えられる。しかしながら、上記のように停電を引き起こす可能性が出てくると、社会的混乱を招くためのテロ攻撃の対象として取り上げられる恐れがある。

つまり、今まで、ホームネットワークは攻撃者から「お目こぼしにあっている状況」にあったに過ぎず、一度目を向けられればかなりの被害が出る状況にあるのではないかと筆者は考えている。セキュリティはホームネットワーク分野における喫緊の最重要課題のひとつであるといえる。

3 章 オープンソース ECHONET Lite 開発環境 echowand

前年度報告書や本報告書の中にも ECHONET Lite[8]に関する言及箇所は多く、少なくとも我が国においてはエネルギーマネジメントを中心とした制御系のホームネットワークにとっては欠かせない規格となっている。ECHONET Lite の規格自体は ECHONET コンソーシアムの Web サイト[9]から取得が可能であるが、手元で試作したり実験するのに用いることのできるオープンソースの ECHONET Lite プロトコルスイートは ECHONET コンソーシアムからは提供されていない。本章では、筆者らが開発し、実験環境で日常的に利用している実装である echowand[10]について述べる。

3.1 echowand と本章の位置づけ

ECHONET Lite デバイスは ECHONET Lite の通信部分と実際のセンサやアクチュエータといった物理的な部分からなっている。実際のセンサやアクチュエータについてはデバイスによって様々な物が利用されることが想定される。一方、ソフトウェア部分は共通のプロトコルを用いた通信部分からなるため、様々なデバイスで共有可能である。

本章では、ECHONET Lite を実装したライブラリの一つである echowand について説明する。echowand を用いることで、ECHONET Lite を用いた ECHONET Lite ノードや ECHONET Lite コントローラの作成が容易となる。echowand を用いて実際の ECHONET Lite ノードを構築するためには、センサやアクチュエータを ECHONET Lite ノードに実装し連携動作するソフトウェアを開発する必要がある。本章ではソフトウェアのみからなる仮想機器オブジェクトを持つ ECHONET Lite ノードの構築方法についての説明も行なう。

本章で作成する仮想機器オブジェクトは、echowand で JavaScript を利用できるようにしたアプリケーションを用いて作成され、JavaScript ファイルを変更することで仮想機器オブジェクトの生成や動作のカスタマイズが可能となっている。読み込む JavaScript ファイルを指定するのは実行時に指定するため、Java の再コンパイルなしで様々な仮想機器オ

プロジェクトをサポートすることが可能である。また動作をカスタマイズするためのオブジェクトを作成し登録することも可能である。

3.2 echowand

echowand は ECHONET Lite の通信部分を Java で実装し、ECHONET Lite のプログラム作成支援を行なうライブラリである。echowand は以下の特徴を持っている。

- Java SE 6 標準ライブラリを用いた実装
- IPv4 および IPv6 の UDP をサポート
- 通信モジュールを入れ替えることで IP 以外のサポートも可能
 - プログラム内でのみ有効な InternalSubnet を実験的に実装
- モジュール化されており、実装したい内容に合わせて様々なインタフェースを提供
 - フレームの送受信を行なうためのインタフェースを提供
 - ECHONET Lite サービスを利用するためのインタフェースを提供
 - ECHONET Lite オブジェクトを抽象化したインタフェースを提供
 - 機器オブジェクト実装インタフェースと機能を提供
 - コントローラ実装インタフェースと機能を提供
- ECHONET Lite デバイスへの Set や Get を可能とした GUI アプリケーションを添付
- ECHONET Lite v1.1 から導入された TCP を試験的にサポート

3.2.1 全体構成

echowand は複数のモジュールから構成されている。主なモジュールは以下の通りである。本節では各モジュールについて説明する。

- common モジュール

- echowand.common パッケージ内で実装
- info モジュール
 - echowand.info パッケージ内で実装
- subnet モジュール
 - echowand.subnet パッケージ内で実装
- transaction モジュール
 - echowand.login パッケージ内で実装
- object モジュール
 - echowand.object パッケージ内で実装
- service モジュール
 - echowand.service パッケージ内で実装

echowand は、モジュール間の依存関係にサイクルが発生しないように設計されている。例えば、service モジュールは全てのモジュールに依存するが、他のモジュールは service モジュールに依存しない構成になっている。一方、common モジュールは他のモジュールから利用されるが、他のモジュールには依存しない設計となっている。モジュール間の依存関係は以下の通りである。

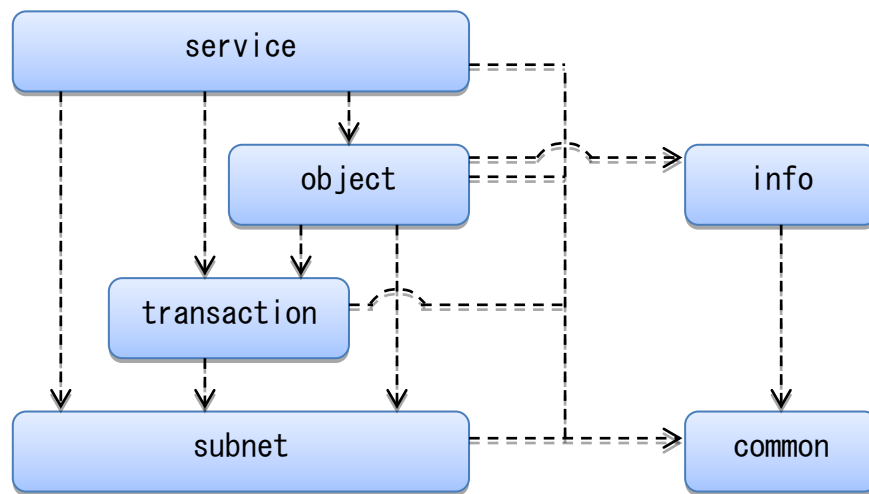


図 3.1 モジュール間依存関係

3.2.1.1 common モジュール

EOJ(機器オブジェクト)、ClassEOJ、EPC(ECHONET プロパティ)、ESV(ECHONET Lite サービス)、Data、PropertyMap 等、他のモジュールから利用される一般的なクラスを提供する

3.2.1.2 info モジュール

ECHONET Lite オブジェクトの構成に関する情報を管理するクラス群を提供する。
LocalObject を生成する際に利用される。

3.2.1.3 subnet モジュール

本モジュールでは、フレームの送受信を行なうクラス群を提供する。下位層プロトコルの実装に依存しないインタフェースを提供する Node クラスと Subnet クラスを用いる事で、下位層のプロトコルの抽象化を行なう。本モジュールを利用することで、ECHONET Lite フレームの送受信を行なうプログラムを簡単に実装できる。主なクラスは以下の通りである。

•Node

ECHONET Lite ネットワークに存在するノードを抽象化したものである。また、マルチキャストの宛先もグループノードとして抽象化を行なっている。ノードに関するアドレス等の情報は、Node によりノード情報の詳細を隠蔽することで他モジュールが下位層のプロトコルに依存するのを防ぐ。

IPv4 および IPv6 を利用した Node として InetNode が実装されている。

•NodeInfo

ECHONET Lite ネットワークのノード情報を抽象化したものである。本クラスは Node クラスとは異なり、Subnet に依存しないノードの情報のみを保持するものとなっている。

•Subnet

ECHONET Lite ネットワークを抽象化したものである。下位層のプロトコルに依存しない Frame の送受信のためのインタフェースを提供する。

IPv4 と IPv6 に対応した Inet4Subnet、Inet6Subnet が実装されている。

•Frame

ECHONET Lite のフレームを抽象化したものである。CommonFrame と、送信ノードおよび受信ノードの Node を保持する。送信ノードと受信ノードの Node を持っていることで、フレームの送信者と受信者を認識することが可能である。

•CommonFrame

ECHONET Lite の電文を抽象化したクラスである。

3.2.1.4 transaction モジュール

Set、Get 等の ECHONET Lite サービスを抽象化するモジュールである。スレッドを用いて MainLoop を実行するという利用方法を想定して作成されているが、MainLoop 相当の処理を別途実装することも可能である。主なクラスは以下の通りである。

- Transaction

単一の ECHONET Lite サービスを開始しフレームの送受信を行うクラスである。ECHONET Lite サービスの詳細設定は TransactionConfig により与えられる。一般的には TransactionManager の createTransaction メソッドを用いて生成される。execute メソッドを呼び出すことで ECHONET Lite サービスを開始する。ECHONET Lite サービス開始後、Subnet を利用することで ECHONET Lite フレームの送信を行い。フレームの送信後、タイムアウト時間まで ECHONET Lite フレームの受信を待つ。

- TransactionConfig

ECHONET Lite サービスの詳細について定義を行うクラスである。送信ノードや宛先ノードの Node や EOJ を指定する。また Set および Get のためのプロパティに関する情報の指定を行う。TransactionConfig は、Transaction を作成する時に情報として渡される。

- TransactionListener

Transaction の開始時、終了時、フレームの受信時の処理を記述するインタフェースである。Transaction クラスは受信フレームの適切なフィルターを行うが、受信フレームを直接処理しない。そこで、本インタフェースを実装したクラスを別途準備して Transaction に登録しておくことで受信フレームの処理を行う。

- TransactionManager

Transaction の管理を行なうクラスである。Listener インタフェースを実装しており、MainLoop

に登録して利用される。**MainLoop** がフレームを受信するたびに、受信フレームの処理を要求される。処理を要求された受信フレームの **TID** を確認し、管理下にある **Transaction** に対する受信フレームであると認識した時には、その **Transaction** に受信フレームを渡す。

•RequestDispatcher

Listener クラスを継承したクラスである。**MainLoop** から要求されたフレームの処理を実現する。フレームの **ECHONET Lite** サービスについて確認を行い、**RequestProcessor** に処理を割り振る。

•RequestProcessor

ECHONET Lite サービスの要求を実行する。インタフェースとなっているため、実装を行ったクラスを定義する必要がある。**object** モジュールの **SetGetRequestProcessor** および **AnnounceRequestProcessor** は本インタフェースを実装している。これにより、**object** への **Set/Get** 及び状態アナウンスサービスの処理を行う。

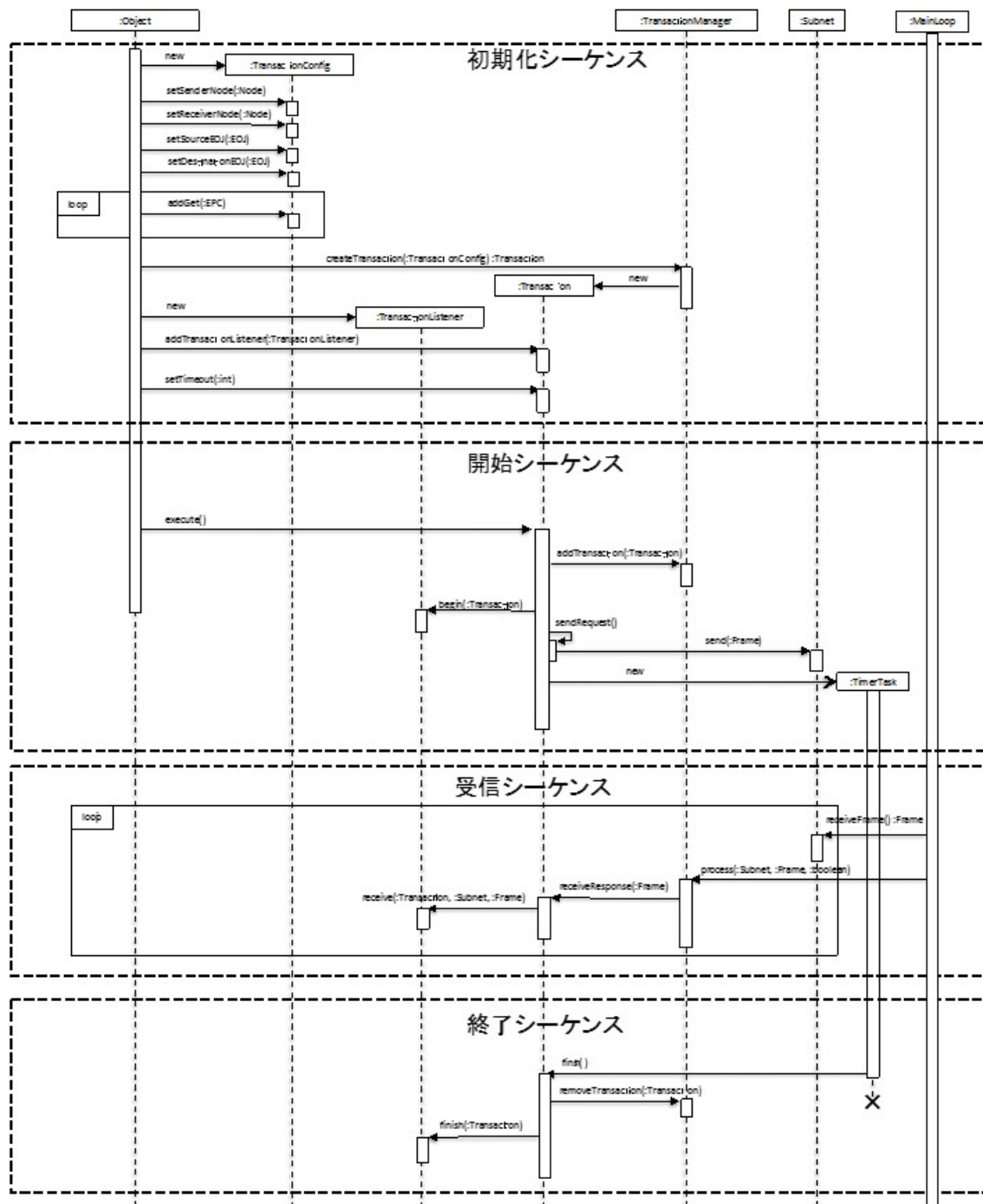
•Listener

MainLoop が利用するインタフェースである。**Listener** インタフェースを実装したクラスを作成し、**MainLoop** に登録して利用される。現在のところ、**ECHONET Lite** サービスの要求フレームを受信するための **RequestDispatcher** クラスと **ECHONET Lite** サービスの応答フレームを受信するための **TransactionManager** の二つが **Listener** を実装している。

•MainLoop

Subnet からフレームを受信し、登録されている各 **Listener** に処理を要求する。**Runnable** を実装しており、独立したスレッドとして動作させることが可能である。**MainLoop** はフレームを受信し、登録されている各 **Listener** に受信したフレームの処理を要求する

Transaction を用いた **ECHONET Lite** サービス開始側の主な処理をシーケンス図で表すと以下のようになる。ここでは、**Get** サービスを利用するものと仮定している。



3.2.1.5 object モジュール

- LocalObject

ローカルな ECHONET Lite オブジェクトを表すクラスである。Set、Get、状態アナウンス可能なプロパティの情報や、その値の保持を行う。また LocalObjectDelegate を複数保持することが可能であり、Set や Get 要求時、および Set の値変更時の処理のカスタマイズが行われる。

- LocalObjectDelegate

ローカルな ECHONET Lite オブジェクトの動作のカスタマイズを行うクラスである。

- LocalObjectManager

LocalObject の管理を行う。ローカルで管理を行うノードプロファイルオブジェクトや機器オブジェクトの登録を行う。

- RemoteObject

リモートの ECHONET Lite オブジェクトを抽象化するオブジェクトである。本オブジェクトに対して Get や Set を行うと、Transaction を利用した ECHONET Lite サービス要求が実行される。また状態アナウンス時の処理を行いたい場合には RemoteObjectObserver を利用する。

- RemoteObjectObserver

状態アナウンス時の処理をカスタマイズするために利用するクラスである。

- RemoteObjectManager

RemoteObject の管理を行う。本クラスのオブジェクトに対して RemoteObject の登録

および検索を行う。また、`InstanceListRequestExecutor` クラスを用いることで、マルチキャストを用いたリモートノードの発見と登録を自動的に行うことも可能である。

3.2.1.6 service モジュール

`echowand` 利用者に簡潔な利用インタフェースを提供するモジュールである。本モジュールを利用することで、`echowand` の初期化から利用までの実装を容易に行うことが可能となる。本モジュールの主なクラスは以下の通りである。

- Core

`Subnet` の生成と初期化、`transaction` モジュールの初期化と `MainLoop` の生成、`object` モジュールの初期化を行うことで、`echowand` としての機能を利用できる環境を整えるクラスである。利用する `Subnet` は標準では `Inet4Subnet` であるが、利用する `Subnet` は `Core` の生成時に指定することが可能である。

- Service

`Core` をラップするクラスである。`Service` クラスは `doGet`、`doSet`、`doObserve` および `doCapture` メソッドを提供している。これらのメソッドは `transaction` モジュールや `Listener` の機能をより簡単に利用できるようにしたインタフェースを提供する。

- LocalObjectConfig

`Core` を用いた `LocalObject` の設定用クラスである。

3.2.2 拡張ライブラリとアプリケーション

`echowand` を基盤として、現在までに以下 4 つの拡張ライブラリ及びアプリケーションが構築されている。ここでは、それぞれのライブラリとアプリケーションについて簡単に

説明する。Scripting アプリケーションを用いた機器オブジェクトの生成方法については、次章で詳細に説明する。

- Harmony

ユーザフレンドリな名前や値を利用した echowand へのアクセスを可能とするライブラリである。コマンドラインから呼び出すアプリケーション等を提供する。

また、リモートの ECHONET Lite オブジェクトの値の変化時に処理を行うプログラミングモデルをサポートしたクラス群も提供している。これらのクラス群を利用することで、データ駆動的なプログラムの作成も可能である。

- Scripting

echowand および Harmony に JavaScript を用いたスクリプト機能を追加したアプリケーションである。echowand や Harmony の機能の一部を JavaScript を用いて記述し動作させることが可能となる。

- EchowandFS

FUSE(Filesystem in Userspace)を利用することで、ECHONET Lite のノードやオブジェクト、プロパティをファイルシステムとしてマウント可能としたアプリケーションである。このアプリケーションを利用してファイルシステムをマウントする事で、プロパティの値をファイル経由で Set、Get することが可能となる。本アプリケーションは、Java から FUSE の機能を利用するために、fuse-jna(<https://github.com/EtiennePerot/fuse-jna>)を利用している。

- Ensemble

Scripting アプリケーションを拡張し、echowand と Harmony の機能に加えて、

EchowandFS との連携も追加したものである。EchowandFS では、ノードやオブジェクト、プロパティを表すディレクトリやファイルの追加はファイルシステムを介してしか行なうことが出来なかったが、Ensemble を用いる事で JavaScript プログラムから可能となる。また、マウントしたファイルシステム内に JavaScript ファイルを作成する機能が追加されており、任意のタイミングでファイルを実行させることが可能となっている。

3.2.3 機器オブジェクト作成方法

echowand を用いて機器オブジェクトを生成する方法について説明する。機器オブジェクトの生成の方法として、object モジュールを利用する方法と、service モジュールを利用する方法がある。機器オブジェクト作成で必要となる作業はどちらもほぼ同じである。ここでは service モジュールを利用した機器オブジェクトの生成について説明する。必要な作業は以下の通りである。

① 機器オブジェクトの設定情報の登録

必要に応じて echowand.info.DeviceObjectInfo を生成し、Get、Set、状態アナウンスの指定とともに EPC の登録を行なう。もちろん、echowand.info.DeviceObjectInfo のサブクラスも作成して実装することも可能である。

② 機器オブジェクトの構築情報オブジェクトの作成

echowand.service.LocalObjectConfig のインスタンスを作成する。この時、先に作成した echowand.info.DeviceObjectInfo を設定情報オブジェクトとして渡す。

③ 動作をカスタマイズするデリゲートオブジェクトの登録

基本的には、echowand が内部的にプロパティのデータを管理し、Set や Get などにより値を書き換えるといった動作を行なう。しかしながら、プロパティの値の変更を即時に

反映させた動作を行なうといった動作や、プロパティ値を `Get` するたびに計算した結果を返すといった動作は不可能である。そこで、このような動作について記述を行なったデリゲートオブジェクト利用することで、このような処理の実現を行なう。

④ 機器オブジェクト構築情報を `Core` に登録

先に作成した `echowand.service.LocalObjectConfig` のインスタンスを機器オブジェクト構築情報として `Core` に登録する。

⑤ `Core` にサービスの開始を要求

`Core` がサービスを開始する時、登録された機器オブジェクト設定情報に基づきローカルの機器オブジェクトが生成される。

3.3 仮想オブジェクト生成

本節では `echowand` を利用した仮想機器オブジェクト作成方法を説明する。前節で説明した通り、`echowand` を利用した機器オブジェクトの生成方法は、`object` モジュールを利用する方法と `service` モジュールを利用する方法がある。ここでは `service` モジュールを利用した方法について説明する。また `Scripting` アプリケーションを利用し、仮想機器オブジェクトの設定と動作のプログラミングは `JavaScript` で行うこととした。これにより、柔軟な仮想機器オブジェクトの構築が再コンパイルなしで可能となる。

3.3.1 実行環境

プログラムの動作確認は `Oracle Java SE 8` で行なった。`Scripting` アプリケーションが利用する `JavaScript` 等のスクリプト言語の機能は `Java SE 6` 以降で対応しているが、`Oracle Java SE 8` は `JavaScript` エンジンの実装が `Rhino` から `Nashorn` に変更されており、以前のバージョンとは非互換な部分があるため注意が必要である。

3.3.1.1 プログラム構成

本節で動作させるアプリケーションは、echowand ライブラリ群と Scripting アプリケーション、ECHONET Lite オブジェクトの詳細について記述した機器設定 JavaScript プログラムから構成される。機器設定 JavaScript プログラムは、作成したい ECHONET Lite オブジェクトに合わせて作成や拡張を行う。

- ・ echowand ライブラリ群

echowand.jar、harmony.jar からなる ECHONET Lite の基本部分を実装したライブラリ群である。本ライブラリを利用することで、ECHONET Lite のデバイスを構築することが可能である。また、コントローラの実装を行なうことも可能である。

- ・ Scripting アプリケーション

scripting.jar からなるアプリケーションである。echowand ライブラリの基本機能を JavaScript から利用可能としたアプリケーションである。デバイスの実装やコントローラの実装を JavaScript で行なうことが可能である。これにより、Java プログラムを再コンパイルすることなく ECHONET Lite デバイスやコントローラ機能の追加が可能となる。また、拡張ライブラリである Harmony の機能も JavaScript から利用可能となっている。

- ・ 機器設定 JavaScript プログラム

仮想機器オブジェクトの詳細を規定した JavaScript プログラムである。本プログラムを読み込む事で様々な機器オブジェクトを模擬した ECHONET Lite デバイスを構築可能となる。本ファイルでは、作成する機器オブジェクトの設定を行なう。また機器オブジェクトの動作をカスタマイズする場合には、デリゲートオブジェクトを生成し利用する。本ドキュメントでは、本ファイルの記述方法について詳細に述べる。

機器設定 JavaScript プログラムとプログラムの関係は下図の通りである。

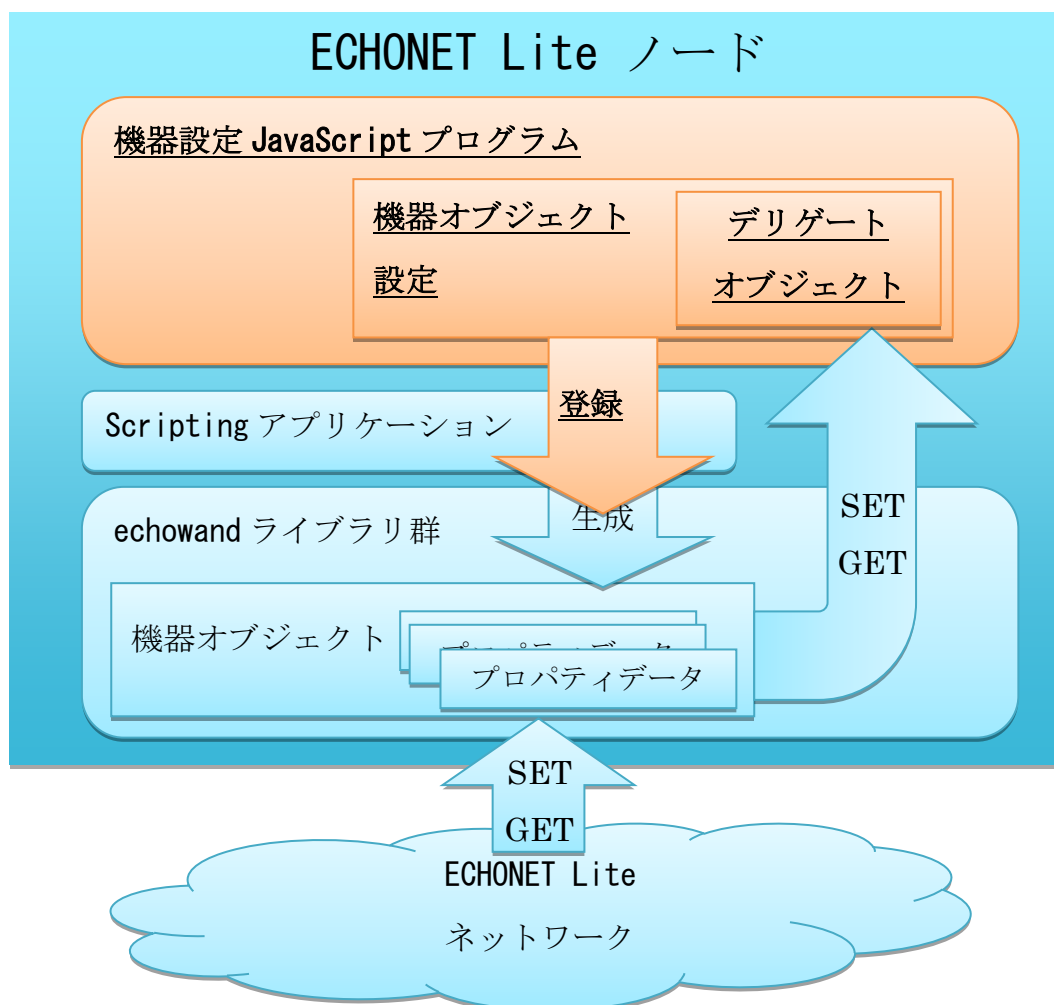


図 3.3 機器設定 JavaScript プログラムとの関係

3.3.1.2 プログラム配置

標準で準備されている jar ファイルを任意の場所($\{ECHODIR\}$)に以下の通り配置する。

利用する jar ファイルや配置場所等についてはプログラムのビルド設定に依存する。

$\{ECHODIR\}/scripting.jar$

`${ECHODIR}/lib/echowand.jar`

`${ECHODIR}/lib/harmony.jar`

利用する仮想デバイス定義プログラム(`device.js` という名前であると仮定する)は実行時に選択可能である。ここでは`${ECHODIR}`の下に置いてあると仮定して説明する。

`${ECHODIR}/device.js`

3.3.2 実行方法

仮想機器オブジェクト定義プログラムを読み込ませて **Scripting** アプリケーションの実行を行う方法は以下の通りである。実行マシンは仮想機器オブジェクトを持つ **ECHONET Lite** ノードとして動作をする。ここでは **Java** の実行方法は、**Linux** を用いると想定している。**Windows** 等、他の **OS** で実行するにはパスの指定方法等が異なる可能性がある。

```
$ java -jar ${ECHODIR}/scripting.jar -i eth0 -c ${ECHODIR}/device.js
```

- `-i eth0`

ECHONET Lite で利用するインタフェースを指定する。この例では `eth0` に設定している。利用するように指定されたインタフェースに **ECHONET Lite** のマルチキャストパケットを送信するようになる。指定しない場合にはデフォルトのインタフェースが利用される。`-i` と指定することで、実行時に対話的に設定することも可能である。

- `-c ${ECHODIR}/device.js`

仮想デバイスの設定を行なった **JavaScript** プログラムを指定する。記述方法については本章で詳細に説明している。仮想エアコンのための完全なサンプルを付録 A に記載する。

3.3.3 機器設定 JavaScript プログラム記述

仮想デバイスとして、仮想エアコンオブジェクトを想定し、機器設定 JavaScript プログラムの記述方法の説明を行う。ここで作成する仮想エアコンオブジェクトは内部的な動作は何も行わず、自発的な動作はしないものである。機器設定 JavaScript プログラムからは echowand ライブラリ群が提供する機能を利用することが可能であるが、ここでは仮想デバイスを作成するために必要最低限の機能の説明にとどめる。仮想エアコンのための完全なサンプルは付録 A に記載する。

3.3.3.1 ECHONET Lite オブジェクト記述用ヘルパーオブジェクト

機器設定 JavaScript プログラムから echowand 中のいくつかの機能を簡単に利用するために、ConfigHelper というオブジェクトが提供されている。本オブジェクトの主なメソッドは以下の通りである。

- ConfigHelper.open

引数で指定された JavaScript オブジェクト内のメソッドをグローバルな環境に展開する。例えば、ConfigHelper.open(ConfigHelper)を呼ぶことで、ConfigHelper 中のメソッドをグローバルな環境に展開する。これにより ConfigHelper のメンバとなっている関数を直接呼び出すことが可能となる。

- ConfigHelper.println

引数で指定されたオブジェクトを文字列で出力する。最後に改行を出力する。

- ConfigHelper.print

引数で指定されたオブジェクトを文字列で出力する。

- `ConfigHelper.printf`

引数で指定されたオブジェクトをフォーマットに合わせて出力する。`Java` の `String.format` を利用して実装されている。

- `ConfigHelper.eoj`

引数としてクラスグループコード、クラスコード、インスタンスコードを文字列で与えることで(e.g. “0ef001”)、`common` モジュールの `EOJ` オブジェクトを生成する。

- `ConfigHelper.ceoj`

引数としてクラスグループコード、クラスコードを文字列で与えることで(e.g. “0ef0”)、`common` モジュールの `ClassEOJ` オブジェクトを生成する。

- `ConfigHelper.epc`

引数としてプロパティコードを数値で与えると、`common` モジュールの適切な `EPC` オブジェクトを返す。

3.3.3.2 ECHONET Lite オブジェクトの設定記述

`echowand` ライブラリの `echowand.info.DeviceObjectInfo` クラスを利用して、作成する `ECHONET Lite` オブジェクトのクラスグループコード、クラスコード、プロパティ、プロパティの初期値、プロパティのアクセスルールに関する情報を設定する。ここで利用する主なオブジェクトとメソッドは以下の通りである。

- `echowand.info.DeviceObjectInfo.setClassEOJ`

生成する仮想デバイスにクラスグループとクラスを設定するメソッドである。インスタンスコードは ECHONET Lite オブジェクト生成時に自動的に割り当てられる。

- `echowand.info.DeviceObjectInfo.add`

本メソッドにより生成する仮想デバイスにプロパティを追加する。第 1 引数で追加するプロパティの EPC を指定する。第 2～4 引数により、`get`、`set`、状態アナウンスの設定を行なう。第 5 引数としては数値配列か数値を与える。数値配列が与えられた場合は、データの初期値として利用される。引数の値が数値の場合にはバイト数となりデータは 0 で初期化される。第 6 引数で保存されるデータフォーマットの制約を行なう。例で利用されている制約オブジェクトは以下の通りである。特に指定しなくても動作に支障はないため、その他の指定方法についてはここでは詳しく説明はしない。

以下、エアコンオブジェクトの構成設定の例を示す。

最初に `echowand.info.DeviceObjectInfo` を作成する。

```
var info = new Packages.echowand.info.DeviceObjectInfo();
```

クラスグループコードとクラスコードの登録を行う。

```
info.setClassEOJ(ConfigHelper.ceoj("0130"));
```

その後、必要なプロパティを登録する。ここでは `OperationStatus(EPC: 0x80)` を `Get`、`Set`、状態アナウンスが有効であるプロパティとして登録する例をあげる。初期値として `[0x31]`を設定し、プロパティ値の制約として `PropertyConstraintOnOff0`を登録している。これにより `0x30` と `0x31` のみが値として許可される。その他の EPC のプロパティも同様

に登録する。

```
info.add(ConfigHelper.epc(0x80), true, true, true, [0x31],  
new Packages.echowand.info.PropertyConstraintOnOff());
```

3.3.3.3 ECHONET Lite オブジェクト動作のカスタマイズ

デリゲートオブジェクトを利用することで、動作のカスタマイズが可能となる。

デリゲートオブジェクトは `echowand.object.LocalObjectDelegate` がスーパークラスとなるようなオブジェクトとして作成する。オーバーライドする必要があるメソッドの第一引数は、動作をカスタマイズするのに必要な情報を設定する **State** と呼ばれるオブジェクトである。

一つの **ECHONET Lite** オブジェクトに複数のデリゲートオブジェクトを登録することが可能である。この場合は、登録された順番でデリゲートオブジェクトのメソッドが呼ばれるようになる。この時、第一引数で与えられる **State** オブジェクトは同一の物が利用される。

デリゲートオブジェクトでオーバーライドが必要となるメソッドと引数の型は以下の通りである。

- `getData(state, object, epc);`

`LocalObject` に `Get` が要求された場合に呼び出されるメソッドである。返す値をカスタマイズする場合には、`state` の `setGetData` メソッドで値を設定する。呼び出される時の引数は以下の通りである。

- `state: echowand.object.LocalObjectDelegate.GetState`
処理結果を保持する
- `object: echowand.object.LocalObject`

対象のオブジェクトを表す

- `epc: echowand.common.EPC`

対象となるプロパティを表す

- `setData(state, object, epc, newData, curData);`

`LocalObject` に `Set` が要求された場合に呼び出されるメソッドである。設定する値をカスタマイズする場合には、`state` の `setSetData` メソッドあるいは `setNewData` メソッドで値を設定する。呼び出される時の引数は以下の通りである。

- `state: echowand.object.LocalObjectDelegate.SetState`

処理結果を保持する

- `object: echowand.object.LocalObject`

対象のオブジェクトを表す

- `epc: echowand.common.EPC`

対象となるプロパティを表す

- `newData: echowand.object.ObjectData`

新規に登録される値を表す

- `curData: echowand.object.ObjectData`

既存の値を表す

- `notifyDataChanged(state, object, epc, curData, oldData);`

`LocalObject` の `Set` 要求後に呼び出されるメソッドである。`setData` メソッド呼び出し時とは異なり、値の変更後に呼び出されることになる。変更前の値と変更後の値は引数として与えられる。呼び出される時の引数は以下の通りである。

- `state: echowand.object.LocalObjectDelegate.NotifyState`

処理結果を保持する

- `object: echowand.object.LocalObject`

対象のオブジェクトを表す

- `epc: echowand.common.EPC`

対象となるプロパティを表す

- `curData: echowand.object.ObjectData`

新たに設定された値を表す

- `oldData: echowand.object.ObjectData`

以前の値を表す

デリゲートオブジェクトでオーバーライドするメソッドの第一引数である `State` オブジェクトについて以下で説明する。このオブジェクトに設定された値を変更することで、`LocalObject` の動作をカスタマイズ可能である。

- `echowand.object.LocalObjectDelegate.GetState`

- `void setDone()`

処理が終了したことを指定する。このメソッドを呼び出すと、残りのデリゲートオブジェクトのメソッドは呼ばれない。

- `void isFail()`

処理に失敗したことを設定する。

- `void setFail()`

処理が失敗した場合に呼び出す。ECHONET Lite の `Get` サービスにより呼び出された場合には、メッセージの応答として `Get_SNA` が返される。

- `void setGetData(ObjectData data)`

`Get` 要求への返答となるオブジェクトを設定する。

- `ObjectData getGetData()`

Get 要求への返答となる予定の値を取得する。この値は `setGetData` メソッドにより上書きされる。

- `echowand.object.LocalObjectDelegate.SetState`

- `void setDone()`

処理が終了したことを指定する。このメソッドを呼び出すと、残りのデリゲートオブジェクトのメソッドは呼ばれない。

- `void isFail()`

処理が失敗したかどうかを返す。

- `void setFail()`

処理に失敗したことを設定する。ECHONET Lite の Set サービスにより呼び出された場合には、メッセージの応答として `Set_SNA` が返される。

- `void setSetData(ObjectData newData, ObjectData curData)`

Set 要求で設定される値を設定する。`newData` には新規に設定される値、`curData` には現在設定される値を指定する。

- `void setNewData(ObjectData data)`

Set 要求で新たに設定される値を指定する。

- `ObjectData getNewData()`

Set 要求で新たに設定される予定である値を指定する。この値は `setSetData` や `setNewData` メソッドにより上書きされる。

- `void setCurrentData(ObjectData data)`

上書きされる現在の値を指定する。

- `ObjectData getCurrentData()`

上書きされる現在の値を取得する。

- `echowand.object.LocalObjectDelegate.NotifyState`

- `void setDone()`

処理が終了したことを指定する。このメソッドを呼び出すと、残りのデリゲートオブジェクトのメソッドは呼ばれない。

- `void isFail()`

処理が失敗したかどうかを返す。

- `void setFail()`

処理に失敗したことを設定する。

3.3.3.4 動作カスタマイズ例

ここでは、日時プロパティ(0x97 と 0x98)のカスタマイズを行うデリゲートオブジェクトの生成及び、Get、Set、値変化時に、メッセージを表示するデリゲートオブジェクトの作成を行う。また、Get をカスタマイズするデリゲートの例として、外気温の設定を可能とするカスタマイズを行なう。このデリゲートを利用する事で 0xF0 に設定した値が外気温として利用されるようになる。

- 日時プロパティのカスタマイズを行うデリゲートオブジェクト

既に準備されている `echowand.object.LocalObjectDateTimeDelegate` を利用する。以下のようにデリゲートオブジェクトを生成する。

```
var dateDelegate=new Packages.echowand.object.LocalObjectDateTimeDelegate();
```

- メッセージ表示のカスタマイズを行うデリゲートオブジェクト

今回メッセージ表示のために作成したデリゲートオブジェクトは以下の通り作成を行った。GetState の `getGetData` や SetState の `getNewData` を呼び出し、その値を `println` で表示している。

```

var showDelegate = new Packages.echowand.object.LocalObjectDelegate({

    getData: function(state, object, epc) {

        println("GET: " + object.getEOJ() + " " + epc + ": " + state.getGetData());

    },

    setData: function(state, object, epc, newData, curData) {

        println("SET: " + object.getEOJ() + " " + epc + ": " + state.getNewData());

    },

    notifyDataChanged: function(state, object, epc, curData, oldData) {

        println("CHANGED: " + object.getEOJ() + " " + epc + ": " + curData);

    }

});

```

- ・ 外気温設定のためのデリゲートオブジェクト

EPC が 0xBE へのプロパティの get 要求の時には、EPC が 0xF0 に設定されている値を利用するように動作を行う。Get 要求の応答に利用するデータは state の setGetData を用いることで設定される。今回外気温設定のためのデリゲートオブジェクトは以下の通り作成を行った。

```

var tempDelegate = new Packages.echowand.object.LocalObjectDelegate({

    getData: function(state, object, epc) {

        if (epc == ConfigHelper.epc(0xBE)) {

            state.setGetData(object.getInternalData(ConfigHelper.epc(0xF0)));

        }

    },

});

```

```

    setData: function(state, object, epc, newData, curData) {

        // do nothing

    },

    notifyDataChanged: function(state, object, epc, curData, oldData) {

        // do nothing

    }

});

```

3.3.4 ECHONET Lite オブジェクトの登録

echowand.service.LocalObjectConfig のインスタンスの作成と、デリゲートオブジェクトの追加を行う。その後、Core の addLocalObjectConfig メソッドを利用することで LocalObjectConfig の登録を行う。Core オブジェクトは echowand の service モジュールで定義されたクラスのオブジェクトである。Core 内で登録された内容に基づいて ECHONET Lite オブジェクトを抽象化した LocalObject が生成される。実際の LocalObject 生成は Core オブジェクトの startService メソッド呼び出し時に行われるが、Scripting アプリケーションが startService メソッドを自動的に呼び出すため、ここでは特に気にする必要はない。

```

var config = new Packages.echowand.service.LocalObjectConfig(info);

config.addDelegate(dateDelegate);

config.addDelegate(tempDelegate);

config.addDelegate(showDelegate);

Core.addLocalObjectConfig(config);

```

3.3.5 実行例

実行例を以下に示す。

・インタフェースをインタラクティブに指定するようにしてコマンドの実行を開始している(1～5 行目)。その後、Scripting アプリケーションのコマンドライン実行機能を用いて、エアコンが登録されていることを確認している(6～8 行目)。外部から `get` や `set` 要求があった場合にデリゲートオブジェクトによるメッセージの表示がされていることが確認できる(9～18 行目)。0x80 への `get` 要求があった場合(9 行目、13 行目)、`set` 要求があった場合(10～12 行目)のメッセージが確認できる。現在の実装では `set` 要求時には最初に `LocalObject` に現在の値の要求を要求するため、`GET` 行が最初に表示されるようになっている(10 行目)。また、EPC が 0xF0 のプロパティに 0x16 を `set` すると、外気温設定のためのデリゲートオブジェクトにより EPC が 0xBE の値が 0x16 に変化していることが分かる(14～18 行目)

3.4 まとめ

本章では ECHONET Lite ライブラリである `echowand` を用いた仮想機器オブジェクトの作成方法と、ECHONET Lite ノードとして動作させるための実行方法について説明した。

`echowand` はモジュールに分割された構成をしており、実装を行いたいアプリケーションに応じて利用するモジュールを選択することが可能である。また、`subnet` モジュールにより下位層プロトコルの詳細は隠蔽されているため、新たな下位層プロトコルへの対応も柔軟に行うことが可能となっている。`echowand` は、機器オブジェクトの生成とコントローラの実装の両方の利用を想定して設計されている。

仮想機器オブジェクトの例として仮想エアコンを取り上げ `echowand` を用いた ECHONET Lite ノードの構築方法について説明した。本ドキュメントでは `echowand` の機能を JavaScript か

ら利用可能とした **scripting** アプリケーションを利用した。これにより、コンパイル作業を行うことなく様々な仮想デバイスを構築することが可能となっている。また、**JavaScript** による仮想デバイス構築方法は、**echowand** を直接利用した **Java** プログラムを行なう場合と必要な作業はほぼ同じであるため、本ドキュメントで説明した仮想機器オブジェクト構築方法は **echowand** を直接利用する場合にも参考となる。

```
1 $ java -jar scripting.jar -i - -c device.js
2 0) default
3 1) eth0
4 2) lo0
5 Select network interface(0-2): 1
6 > var objects = Service.getLocalObjectManager().getDeviceObjects();
7 > objects.get(0);
8 var it = LocalObject{E0J: 013001} : LocalObject
9 > GET: 013001 x80: 31
10 GET: 013001 x80: 31
11 SET: 013001 x80: 30
12 CHANGED: 013001 x80: 30
13 GET: 013001 x80: 30
14 GET: 013001 xBE: 00
15 GET: 013001 xF0: 00
16 SET: 013001 xF0: 16
17 CHANGED: 013001 xF0: 16
18 GET: 013001 xBE: 16
```

付録 A 機器設定 JavaScript プログラムサンプル

例として作成した仮想エアコンを記述した **device.js** の全プログラムは以下の通りである。

/*

* **ConfigHelper** のメソッドをグローバルに展開

```

*/

ConfigHelper.open(ConfigHelper);

/*

 * エアコンオブジェクトの設定

*/

var info = new Packages.echowand.info.DeviceObjectInfo();

info.setClassEOJ(cej("0130"));

info.add(epc(0x80), true, true, true, [0x31],

        new Packages.echowand.info.PropertyConstraintOnOff());

info.add(epc(0x83), true, false, false, 9);

info.add(epc(0x84), true, false, false, 2);

info.add(epc(0x85), true, false, false, 4);

info.add(epc(0x86), true, false, false, 4);

info.add(epc(0x87), true, true, false, [0x64],

        new Packages.echowand.util.ConstraintByte(0, 100));

info.add(epc(0x89), true, false, false, 2);

info.add(epc(0x8C), true, false, false,

        [0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41]);

info.add(epc(0x8D), true, false, false,

        [0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41]);

info.add(epc(0x8E), true, false, false, [0x07, 0xdf, 0x01, 0x05]);

```

```

info.add(epc(0x90), true, true, false, [0x41],
        new Packages.echowand.util.ConstraintByte(0x41, 0x44));
info.add(epc(0x92), true, true, false, 2);
info.add(epc(0x94), true, true, false, [0x41],
        new Packages.echowand.util.ConstraintByte(0x41, 0x44));
info.add(epc(0x96), true, true, false, 2);

info.add(epc(0x97), true, false, false, 2);
info.add(epc(0x98), true, false, false, 4);

info.add(epc(0x9A), true, false, false, [0x41, 0x00, 0x00, 0x00]);

info.add(epc(0xA0), true, true, false, [0x41],
        new Packages.echowand.util.ConstraintSize(1));
info.add(epc(0xA3), true, true, false, [0x31],
        new Packages.echowand.info.PropertyConstraintOnOff0);

info.add(epc(0xB0), true, true, true, [0x41],
        new Packages.echowand.util.ConstraintByte(0x40, 0x45));
info.add(epc(0xB1), true, true, false, [0x41],
        new Packages.echowand.util.ConstraintByte(0x41, 0x42));
info.add(epc(0xB2), true, true, false, [0x41],
        new Packages.echowand.util.ConstraintByte(0x41, 0x43));
info.add(epc(0xB3), true, true, false, [0x18],
        new Packages.echowand.util.ConstraintByte(0x00, 0x32));

```

```

info.add(epc(0xBB), true, false, false, [0x18],
        new Packages.echowand.util.ConstraintSize(1));

info.add(epc(0xBE), true, false, false, [0x31],
        new Packages.echowand.info.PropertyConstraintOnOff0);

/*
 * 温度設定用の拡張プロパティを追加
 */

info.add(epc(0xF0), true, true, false, 1);

/*
 * 日時プロパティのカスタマイズを行うデリゲートオブジェクト作成
 */

var dateDelegate = new Packages.echowand.object.LocalObjectDateTimeDelegate();

/*
 * メッセージ表示のカスタマイズを行うデリゲートオブジェクト
 */

var showDelegate = new Packages.echowand.object.LocalObjectDelegate({
    getData: function(state, object, epc) {
        println("GET: " + object.getEOJ() + " " + epc + ": " + state.getGetData());
    },
    setData: function(state, object, epc, newData, curData) {
        println("SET: " + object.getEOJ() + " " + epc + ": " + state.getNewData());
    },

```

```

        notifyDataChanged: function(state, object, epc, curData, oldData) {

            println("CHANGED: " + object.getEOJ() + " " + epc + ": " + curData);

        }

    });

    /*

    *外気温設定のためのデリゲートオブジェクト

    */

    var currentTemp;

    var tempDelegate = new Packages.echowand.object.LocalObjectDelegate({

        getData: function(state, object, epc) {

            if (epc == ConfigHelper.epc(0xBE)) {

                state.setGetData(object.getInternalData(ConfigHelper.epc(0xF0)));

            }

        },

        setData: function(state, object, epc, newData, curData) {

            // do nothing

        },

        notifyDataChanged: function(state, object, epc, curData, oldData) {

            // do nothing

        }

    });

    /*

    * エアコンオブジェクトの登録

```

```
*/  
  
var config = new Packages.echowand.service.LocalObjectConfig(info);  
  
config.addDelegate(dateDelegate);  
  
config.addDelegate(tempDelegate);  
  
config.addDelegate(showDelegate);  
  
Core.addLocalObjectConfig(config);
```

参考文献

- [1]一般社団法人 情報通信技術委員会(TTC) ドキュメントダウンロードページ、
http://www.ttc.or.jp/document_db
- [2]スマート IoT 推進フォーラム Web サイト、
<http://smartiot-forum.jp/>
- [3]IoT 推進コンソーシアム セキュリティ WG Web サイト、
<http://www.iotac.jp/wg/security/>
- [4]内閣サイバーセキュリティセンター IoT セキュリティのための一般的枠組、
http://www.nisc.go.jp/active/kihon/iot_fw2016.html
- [5]総務省・経済産業省 IoT セキュリティガイドライン、
<http://search.e-gov.go.jp/servlet/Public?CLASSNAME=PCMMSTDETAIL&id=145208784%20&Mode=2>
- [6]一般社団法人 重要生活機器連携セキュリティ協議会 Web サイト、
<https://www.ccds.or.jp/>
- [7]技術研究組合 制御システムセキュリティセンター Web サイト、
<http://www.css-center.or.jp/>
- [8]ECHONET コンソーシアム Web サイト、
<https://echonet.jp/>
- [9]ECHONET 規格書ダウンロードページ、
<https://echonet.jp/spec/>
- [10]ECHOWAND 公開ページ、
<https://github.com/ymakino/echowand>