2004年度 修士論文

RESTを用いた家電制御システム構築へのアプローチ

提出日:2005年2月2日

指導教員：中島達夫教授

早稲田大学大学院 理工学研究科
情報ネットワーク専攻
学籍番号：3603U032-1
生形 裕貴

目 次

図 目 次

表 目 次

概 要

ユビキタスコンピューティングへのコンピュータパラダイムの推移により，様々なデバイスがネットワーク経由で操作可能になるホームコンピューティング環境が実現されつつある．ホームコンピューティング環境では家の中の様々なものが操作の対象となる．そのため，エンドユーザがそれらのデバイスを自由に組み合わせてアプリケーションを構成可能であることが要求される．従って，デバイス間の相互運用性や互換性の向上が課題である．

同様の要件を満たしているシステムとして，WWW が挙げられる．WWW は，*REST*（*Representational State Transfer*）と呼ばれる簡潔なモデルを基幹としており，その成功を成し得た核となる部分は，少数セットコマンドにより実現する簡潔なインタフェースと，統一されたアドレス空間からなる簡潔なアーキテクチャである．WWW はこのモデルにより，高い相互運用性と柔軟な拡張性を実現している．

そこで本論文では，ホームコンピューティング環境における相互運用性の問題に対して，REST を導入し，ホームコンピューティング環境においてもその簡潔なアーキテクチャが有用であることを検証する．具体的には，既存のホームコンピューティングミドルウェアのひとつである SENCHA に REST を導入する．設計の章では，REST で定義されているコンポーネント，コネクタ，データという基本要素を用いて SENCHA を分析し，リソース，リプレゼンテーションという概念を用いて再設計を行う．

REST の導入により，SENCHA は軽量なアーキテクチャを実現することができ，さらに，デバイス間のインタフェースによる依存を最小限に抑えることによって，エンドユーザによる自由なデバイスの構成が可能になった．メディアデータの取り扱いにおける同期など，REST のモデルでは網羅しきれていない部分もある一方で，ホームコンピューティング環境においても，REST のモデルのように簡潔なアーキテクチャを導入することは充分有用であるという結論を得た．

Abstract

By the transition of the computer paradigm from the personal computing to the ubiquitous computing, user surroundings, such as home appliances, audios, furnitures, come to have computational power and network capability. These enhanced devices communicate with each other and support a sophisticated user's life. Such an environment is called a *home computing environment* and in which the requirement for interoperability and compatibility must be fulfilled because there are many devices with various types and functions, and there is a requirement of the user to construct his/her applications freely for his/her needs or tasks.

WWW(World Wide Web) is one of the system that fulfills a similar requirement. The basis of WWW is simple model which is called *REST(Representational State Transfer)*, and the core of REST is consists of 1) genericity of interfaces which is achieved by a few sets of command, 2) uniquely addressable resources. Thus, WWW has achieved a high interoprability and a flexible extensibility by this model.

In this thesis, REST is introduced into the home computing environment to cope with the problem of interoperability, and it is verified that the simple architecture of REST is useful in the home computing environment Concretely, REST is introduced into SENCHA that is the Home-computing middleware based on UPnP(Universal Plug and Play). In the chapter of design, SENCHA is reconstructed by analyzing it by using concept of basic elements defined in REST architectural style; component, connector, and data, and using abstraction of resource and representation.

As for SENCHA, the introduction of REST enables light weight architecture and minimizes dependency of the interface. Thus, flexible composition of the devices by the end users become possible. It can be considered that it is very useful to introduce architecture as simple as the model of REST into the home computing environment while it is not suitable for synchronization in the handling of the media data.

第1章　序論

1.1　背景

情報処理技術の進歩と，コンピュータの高性能化，小型化，低価格化によって，今日におけるコンピュータを使うスタイルも変貌を遂げてきている．かつては，ひとつの巨大なコンピュータを複数の端末から複数ユーザが共有するという，ユーザとコンピュータの関係が他対一のスタイルだったのに対し，今日ではユーザに対して一台のコンピュータというパーソナルコンピューティング (PC) 環境と呼ばれるパラダイムが実現している．そして今，ユーザ一人に対して複数のコンピュータが利用されるというユビキタスコンピューティング環境 [1] のパラダイムに移行しつつある（図 1.1 参照）．

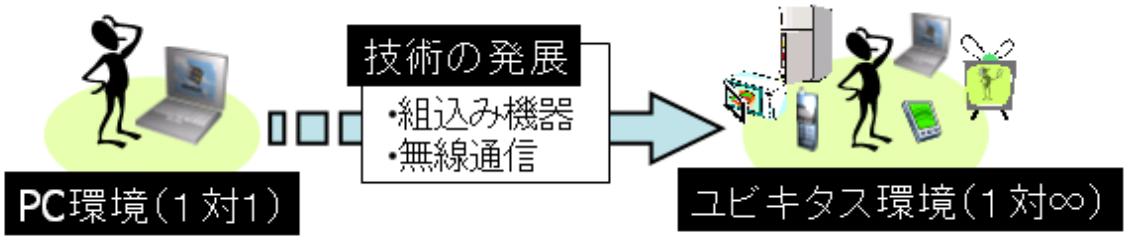

図 1.1: コンピュータ環境の変化

ユビキタスコンピューティング環境は，様々な形態のコンピュータがいたるところに遍在し，いつでもどこでもユーザがコンピュータを利用できる環境である．例えば，白物家電やオーディオのような大型のものから，センサ能力[1]によりコンテキスト情報を提供するイスや机などのスマートオブジェクトと呼ばれる小型のものに至るまで，様々なものが存在する．その中でも，PC 環境のパラダイムとの大きな違いは，ユーザがコンピュータを意識せずにコンピュータによる多様なサービスを受けることができるという点である．

このような環境は，一般家庭内で実現されつつあり，コンピュータを組み込まれたテレビやステレオなどのオーディオ機器，電子レンジや冷蔵庫，エアコンなどのいわゆる情報家電と呼ばれるものも，実際に消費者の手に届くようになってきている．これらは従来の電化製品のように各々が単独で動作するだけではなく，それぞれがワイヤレス LAN や Bluetooth，IEEE1394 などのネットワークを経由して協調動作することができ，今まで以上のサービスを提供することが可能である．

このような情報家電や，スマートオブジェクトなどを対象とした，一般家庭において実現されるユビキタスコンピューティング環境を，我々はホームコンピューティン

[1]圧力，温度，加速度など，ユーザの周りの状況を把握するためのセンサ機能

グ環境と呼んでいる．

1.2 ホームコンピューティング環境のシステム要件

上述の通り，ホームコンピューティング環境では，家電からセンサを組み込まれたモノに至るまで様々なコンピュータが対象となる．さらに，これらのデバイスがネットワークによって接続され，互いに情報をやりとりするということが重要となる．つまり，今まで単独で動作して目的を達成していた家電や日常のモノが，役割を変え，新しいサービスを実現するために互いに協調することが要求されるということである．このように，協調動作することが前提になるホームコンピューティング環境のデバイスは以下のような要件を満たさなくてはならない．

1.2.1 エンドユーザによる開発

ユビキタスコンピューティング環境では，コンピュータはユーザの見えるところから隠れ，ユーザがコンピュータの存在を意識せずにその恩恵（サービス）を受けられるようになるのが，ひとつのゴールであるとされている．しかしその一方で，ユーザがそのサービスを自由に設定したいという要望があることも考慮しなくてはならない [5]．

Mavrommati は，[5] において，エンドユーザプログラミングという概念を提案している．コンピュータによる処理能力や通信能力，センサ能力を得たオブジェクト (ハイパーオブジェクト) を対象とするユビキタスコンピューティング環境を背景としている．Mavrommati は同時に，従来のオブジェクトは，設計者によって意図された使用方法が設計時に決定され，それがオブジェクトの形状や能力に予め埋め込まれているが，ユーザは実際にはそれの範疇を超えた使用法を考え出すことがあると言及している[2]．その対象が，今までのオブジェクトからハイパーオブジェクトへと変化したところで，そのようなユーザの特性は変わらないとしている．

このように，ユーザがオブジェクトの使用法に対して新しい方法を考え出すということはごく自然な流れであり，少なからずそのような要求があるということが言える．このことは，我々の想定しているホームコンピューティング環境でも同じことが言えると考えられる．各ユーザが自由にデバイスを接続したいという要求があると考えるのは，ごく自然なことである．言い換えると，ホームコンピューティング環境におけるデバイスは，使用法を限定したインタフェースを設計時に決めてしまうのではなく，アプリケーションロジックはエンドユーザが使用時に自由に決められるようシンプルかつ柔軟なものにする必要があると言える．

相互運用性

上記のエンドユーザプログラミングの必要性を考察すると，各デバイス間の相互運用性は２つの理由から重要であると考えられる．１つ目の理由としては，ユーザが自

[2]例えば，マグカップの例を引用すると，元来はコーヒーなどの液体を保持するための道具であり，台所と密接な関係を持つという性質を備えている．しかし，一度ユーザが変われば，マグカップはペン立てとして使用され，台所ではなく子供部屋など他の部屋と関係を持つようになる．このように使い方はユーザによって新たに編み出されていくものであるとしている．

由にデバイスを選択し，それらを自由に組み合わせたいという要望があること．２つ目には，ユーザの要求は様々であり，組み合わせやその接続方法は多岐に渡ると想定される点である．

下に示す図 1.2，図 1.3 の例では，監視カメラによって感知される状況の変化というイベントをどのデバイスに送るかを設定した例である． 図 1.2 では，監視カメラのイベントをテレビに発行し，その映像をテレビに出力するという例である．一方，図 1.3 では，監視カメラのイベントをスピーカーに送る例である．後者の場合，スピーカーは映像を表示するためのデバイスではないため，イベントを受け取るとビープ音などによって，そのイベントの存在を知らせるというアプリケーションが考えられる．また，図 1.2 の場合も，上記の接続だけでなく，監視カメラから発行されるイベントを受け取ったらテレビの電源をオフにするというアプリケーションも考えられる．

図 1.2: 接続例１（監視カメラ-テレビ）

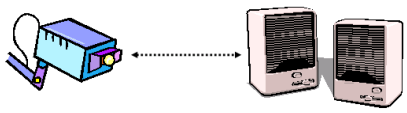

図 1.3: 接続例２（監視カメラ-スピーカ）

このように，全自動であるか手動であるかの手段は別としても，その下層ではデバイスのベンダとは別の第三者が自由にデバイス同士を結合できるメカニズムが必要になってくると考えられる．そのためには，予め考えうる接続のためのインタフェースを想定し，システムを構築するというアプローチをとるのではなく，どんな組み合わせにも対応しうる限りなく柔軟なインタフェースを備えたシステムである必要がある．さらに，そのインタフェースは，アプリケーションのロジックに依存しない柔軟なものである必要がある．

互換性

また，ホームコンピューティング環境においてユーザの操作の対象となるデバイスでは，以下の特徴も考慮しなくてはならない．

- 大型の家具やテレビ，ステレオなどの家電製品は高額なデバイスのため，一度購入・配置されると比較的長い間使用され続ける
- 机やイス含む日常物や家具など，従来の家電製品ではなかったものにも組み込み機器が搭載されることを想定すると，ユビキタスコンピューティング環境下の制御対象となるデバイスの数は膨大な数にのぼる．そのため，個々の機器内のシステムを常にアップデートし，最新の状態を保つことは困難である．

以上から，ホームコンピューティング環境では，世代を超えた新旧デバイス間の連携が必然的に要求されることが分かる．即ち，新旧バージョンでの高い互換性が保証されていることが要求される．

この互換性の問題を考察すると，長期に渡り有効なインタフェースが必要であると考えられる．インタフェースに含まれる依存には，意味的な依存を表す実依存と，プロトコルなどの人工的な依存とに分類できる．

例えば，電化製品を使う場合に，実依存となる部分は電力を必要としているという事実を指す．人工的な依存というのは，電気を得るためのコンセント形状や電圧ということになる．このように，実依存は環境に左右されない根本的な要求のことを指し，人工的な依存とは，満たさなければいけない技術的要件のことを指す．

ホームコンピューティング環境を含む分散システムの場合，必要とするデータを得る（実依存を解決する）ために，プロトコルやデータタイプ，メソッドの名前や引数など（人工的な依存）を解決しなくてはいけない．実依存は，変化することはないが，人工的な依存は仕様の違いや，同じ仕様でもバージョンによって変化してしまうため，いかに長期に渡る使用に耐えうるかは，人工的な依存がどのように解決されるかに大きく依存していると言える．

1.3 WWW における REST アーキテクチャ的スタイル

1.2 節において，ホームコンピューティング環境に必要なシステム要件を挙げた．同様の要件を満たし，実際に稼動しているシステムの例として WWW（World Wide Web）が挙げられる．WWW 内では，様々な組織によって管理されるサーバが存在し，また，その中でも長期に渡り稼動しているサーバもあれば，新規に WWW のネットワークに参入するサーバもある．このように，WWW は多様なノードが存在する環境であり，それらが破綻することなく通信し，かつ，普及後 10 年という年数を経てもその機能を陳腐化させずに存続しているシステムである．

この WWW の根底にあるアーキテクチャの制約をまとめたものが，REST（REpresentational State Transfer）[6][7][8] である．REST は，WWW における相互運用性，拡張性，スケーラビリティなどの特筆すべき特徴を成し得た要素をソフトウェアアーキテクチの視点からまとめ，データ，コンポーネント，コネクションの基本となる要素の定義と，それらの振る舞いに対する制約，そして，それらの関連をまとめたものである．REST の提唱者である Fielding は，このようなアーキテクチャに対する制約のセットをアーキテクチャ的スタイル（Architectural Style）[3]と呼んでいる．

REST は，WWW に特化したモデルではなく，REST のアーキテクチャ的スタイルを実現しているシステムのひとつが WWW という位置付けである．HTTP による簡潔なノード間インタフェースの実現と，URI による一意なリソースの識別という簡潔なアーキテクチャが WWW の成功を成し得た核となる部分である．

1.4 研究の目的

本論文では，前述したホームコンピューティング環境における相互運用性や互換性のシステム要件を実現するための手段として，WWW の根底となるモデルである REST を導入する方法を検証する．

実装は，SENCHA システム [2][3] を REST を適用し再構築する．SENCHA は，ホームコンピューティング環境の実現を目的としたシステムであり，アーキテクチャとしては，UPnP を基盤としたユーザ中心のアーキテクチャを採用し，デバイスコントロールのメカニズムには SOAP（Simple Object Access Protocol）[18] を採用している．

[3]モデルと同義

検証は元々の SENCHA で採用されている SOAP と REST の比較と，定性的な評価によって行う．

1.5 論文の構成

本論の構成は以下の通りである．

第２章 導入

本研究の核となる SENCHA と REST について述べる．SENCHA の説明では，特徴・機能・アーキテクチャの他に，SOAP についても考察する．

REST の説明では，モデルの重要な要素となるコンポーネント・コネクタ・データについての説明と，リソースとリプレゼンテーションという REST の中心的な概念を述べる．

第３章 設計・実装

SENCHA に REST を導入するための分析と設計，実装について述べる．

REST で定義されているコンポーネント・コネクタ・データを元に SENCHA を分析し，リソース，リプレゼンテーションを中心に再構築を行う．

第４章 評価・議論

定量的評価として，REST による実装と，SOAP による実装との比較を行う．定性的評価として，ホームコンピューティング環境の要件を満たしているかを考察する．

第５章 関連研究

SENCHA と類似したアーキテクチャを持つ Jini，ICrafter，SpeakEasy の３つの関連研究について述べる．

第６章 結論

結論，及び今後の展望を述べ，まとめとする．

第2章　導入

本章では，本研究の核となる SENCHA と REST について説明する．

2.1 SENCHA: Smart Environment for Controlling Home Appliances

SENCHA の主な目的は，ネットワーク上に分散するデバイスを管理及び協調させること（デバイスアグリゲーション）にある．SENCHA は，デバイスの発見・管理，ユーザが操作するためのユーザインタフェース（UI）の生成，コントロールの３つの機能によって構成される．

2.1.1 特徴

ユーザ中心のアーキテクチャ

まず，SENCHA の特筆すべき特徴として，ユーザ中心のアーキテクチャを採用しているという点が挙げられる．ホームコンピューティング環境におけるデバイスアグリゲーションを実現しようとする場合，数多くのデバイスを管理する必要がある．そのため，管理するためのシステムを必要とするが，多くの場合，Jini[26] や HAVi[27] のようにホームゲートウェイサーバによってそれを実現するアプローチが主流である．しかし，このアプローチでは，ネットワークセグメントごとにゲートウェイサーバを設置しなくてはならず，コスト的及び技術的な負担をユーザに強いてしまうことになり，かつ，新しくそのサーバに管理される環境に入ってくるユーザは，環境ごとにそのサーバに対応するためのソフトウェアのダウンロードや設定などを強いられてしまうため，柔軟なアプローチとは言い難い．

そこで，SENCHA では，このようなホームゲートウェイサーバをユーザの持ち歩くパーソナルデバイス上に配置するというユーザ中心のアーキテクチャを採用している．これによって，環境ごとにユーザの設定を変更するのではなく，環境が変わっても管理システムを携帯することによって，常に同じ設定を適応できるという利点が得られる．

SENCHA では，パーソナルデバイス上に配置されたシステムのことを PAC（Personal Appliance Coordinator）と呼び，PAC によるコントロールの対象となるデバイス（情報家電，スマートオブジェクト）を総称してスマートアプライアンス，または，アプライアンスと呼び，パーソナルデバイス，アプライアンスをまとめてデバイスと呼ぶ．

パーソナライゼイション

ホームコンピューティング環境では，ユーザの管理の対象となるデバイスは相当数にのぼると想定される．そのため，ユーザが，数多くのデバイスの中から目的とするデバイスを選び出すのは負担になってしまうと考えられる．このユーザビリティの問題に対応する方法として，SENCHA ではパーソナライゼーションの機構を搭載している．具体的には，ユーザの嗜好や，かつての使用履歴などを元に，デバイスをコントロールするためのユーザインタフェース（UI）をそのユーザに適した形にカスタマイズする機能を提供している．

また，個々のユーザに特化したカスタマイズ機能を実現するにあたり，ユーザ中心のアプローチがプライバシ保護の問題にも一役を担っている．パーソナライゼーションゼーションに必要なユーザの情報は，パーソナルデバイス上に保持されているものと想定しており，かつ，デバイスを管理する PAC がパーソナルデバイス上で起動していることにより，ユーザの個人情報を外部に出すことなく，デバイスアグリゲーションに利用できるため，不必要な情報漏洩の回避を実現している．

コンポーネントフレームワーク

SENCHA では，システムの構築･管理の容易性の観点から，システム構築に OSGi(Open Services Gateway initiative)[11] の提供するコンポーネントフレームワーク（以下，OSGi とする）を採用している．OSGi は Java によるコンポーネントフレームワークで，ホームゲートウェイとなるサーバで利用されることを想定されているため，仕様で UPnP（Universal Plug and Play）[12] やサーブレットなどの標準的なサービスも組み込まれている．

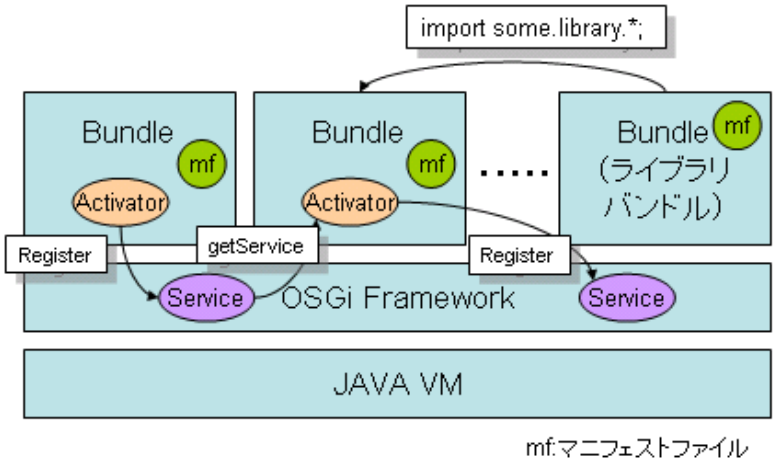

図 2.1: OSGi 概要

OSGi におけるコンポーネントは，jar ファイルによるバンドル[1]として実現されている．また，フレームワークがコンポーネントを活性化させるためのエントリポイントとして，Activator インタフェースが定義されており，このインタフェースを実装

[1]OSGi におけるソフトウェアモジュールはバンドルという単位で管理される

することにより，バンドル内で実装されているサービスの生成，登録を行えるようになる．バンドルのサービスは，Java のインタフェースと，それを実装するオブジェクトによって提供される．バンドル内に Activator インタフェースを実装したクラスが存在する場合，Activator がそのバンドルの提供するサービスを OSGi フレームワークに登録し，または，フレームワークに登録されている他バンドルの提供するサービスの参照を得るという処理をする．Activator が存在しない場合は，そのバンドルはサービスは提供せず，パッケージを提供する単純なライブラリバンドルとして機能する（図 2.1 参照）．

OSGi 内部では，各バンドルが別のスレッドとして起動しているため，個々のサービスのインストール・アンインストール，もしくは，スタート・ストップは動的に行うことができる．特に，サービスの登録・削除に関するイベントなどバンドルレベルの管理機構を OSGi フレームワークが提供しているため，ソフトウェアの管理を効率良く行えることができる．

2.1.2 アーキテクチャ

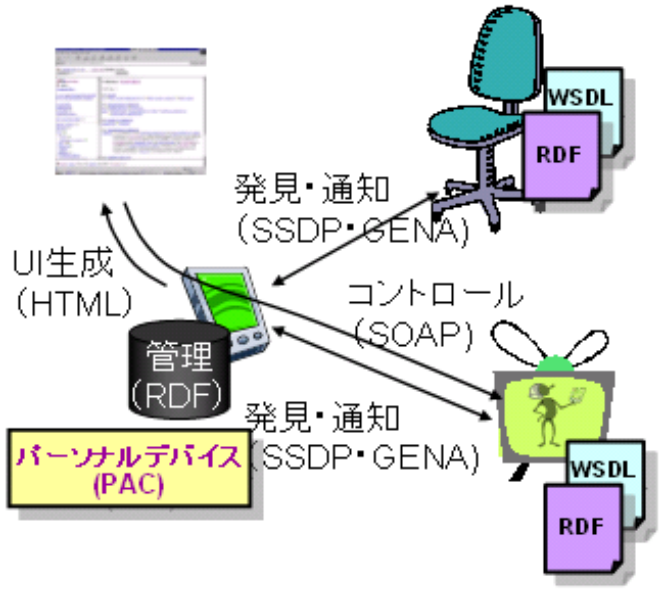

図 2.2: SENCHA 概要図

SENCHA では基本となる技術に UPnP[12] を採用しており，Java と OSGi フレームワークを用いて実装されている．UPnP は，Universal Plug and Play の略で，プラグアンドプレイの概念[2]をネットワークデバイスに対応させたものである．つまり，ネットワーク機器の接続に必要な IP アドレスの設定，各種デバイスドライバのインストールなどの煩雑な設定を自動化する技術である．

図 2.2 に SENCHA の全体像を示す．2.1.1 節で説明した通り，スマートアプライアンスを管理する機構（PAC）はパーソナルデバイス上に配置され，それを中心としたアー

[2]PC 上で USB や PC カードを挿した際，自動で検出し，リソースの割り当てなどの OS の設定と，デバイスドライバのロードを自動で行うもの

キテクチャとなっている．各スマートアプライアンスは RDF(Resource Description Framework)[13] による特性記述ファイル内にデバイスと，そのデバイスの提供するサービスに関する情報を保持している．これらの特性記述を取得するために，SENCHA では，UPnP の仕様で策定されている SSDP（Simple Service Discovery Protocol）[28] を採用している．また，この特性記述内に，アプライアンスが提供するサービスのインタフェースとなる WSDL （Web Services Description Language）[14] ファイルへの参照も含んでいる．PAC は，この WSDL ファイルを参照することによって，動的な SOAP-RPC 呼び出しが可能になっている．

PAC，スマートアプライアンスそれぞれの内部設計を以下の節で説明する．

PAC (Personal Appliance Coordinator)

PAC は，図 2.3 に示されるように５つのコンポーネントから成り立っている．

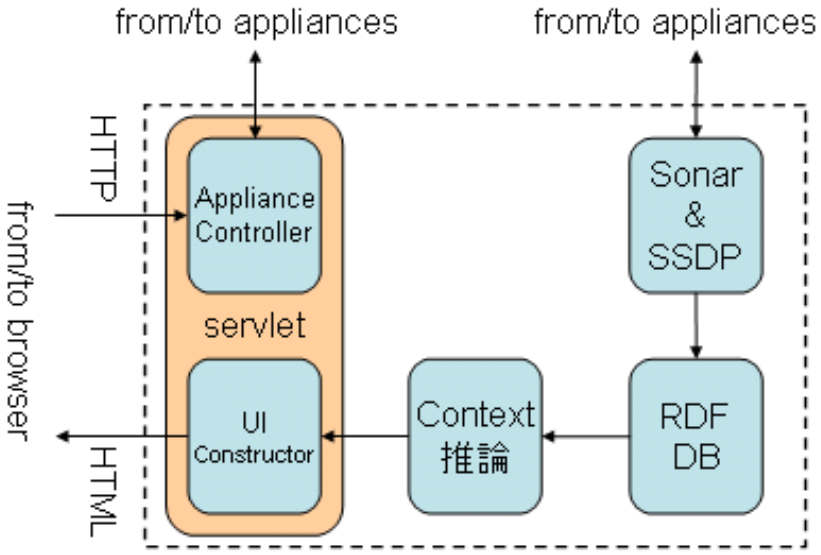

図 2.3: PAC の構造

ディスカバリ まず，ネットワーク上のデバイスを発見するための Sonar コンポーネントがある．これは SSDP のプロトコルスタックを含み，UPnP の仕様に定義されている SSDP のメッセージングを行うことが可能である．SSDP は HTTP を拡張した UDP/IP ベースのプロトコルである HTTPMU（マルチキャスト）と HTTPU(ユニキャスト) 上に構築されたプロトコルである．SSDP では，デバイスがネットワークに参加した際に自分の存在をアドバタイズするための告知（アドバタイズ），検索（サーチ），削除（ディスコネクト）の機能を定義するとともに，メッセージをマルチキャストするためのマルチキャストアドレス（239.255.255.250）とマルチキャストポート（1900）を定義している[3].

[3] このアドレスとポートは IANA（Internet Assigned Number Authority）によって予約されている

Sonar コンポーネントは SSDP に定義された 4 種類のメッセージ（alive, byebye, M-search, response）を駆使し，ネットワーク上のデバイスを発見し，そのデバイスの機能や属性などの情報を取得することができる．UPnP では，デバイス情報は独自のスキームによって定義された XML 記述によって表現されるが，SENCHA では，このデバイス記述に RDF を導入する．RDF を用いた特性記述に関する詳細は 2.1.2 節にて説明する．

UI 生成 Sonar コンポーネントによって発見されたデバイスの特性記述である RDF ファイルは，RDFDB コンポーネントによって取得・管理される．RDFDB コンポーネントは，Hewlett-Packard 社のセマンティック WEB グループが開発した Jena[15] を採用している．Jena の機能としては，RDF を三つ組のステートメントとして処理するだけではなく，永続記憶の機能（ファイルシステム，RDF），オントロジのためのサブシステム，SQL に類似した操作インタフェースなどの機能を提供している．

PAC は，RDFDB 内の情報を用いることによって，UI（ユーザインタフェース）を自動生成するが，このとき，コンテキスト推論コンポーネントを介すことにより，ユーザの過去の履歴や状況，また，好みの情報を利用し，必要な情報に絞り込むことができる．UI コンストラクタでは，HTML による UI を自動生成している．

コントロール ユーザからの入力は自動生成された HTML より Appliance Controller コンポーネントに送られる．このコンポーネントは，UI からの HTTP メッセージを解析し，さらに，アプライアンスの WSDL ファイルを取得し，スタブオブジェクトの生成，SOAP-RPC によるリモートメソッド呼び出しを実行する．

Smart Appliances

アプライアンスは以下の 3 つのコンポーネントによって構成される（図 2.4）．外部に公開されるインタフェースは，Sonar コンポーネントの提供する SSDP のメッセージによるインターフェースと，Apache Axis サーバ [16] によって提供される SOAP インタフェースの二つがある．

スマートアプライアンスでも PAC で使用している Sonar コンポーネントを使用しているが，使用する機能はアドバタイズのみで，他デバイスのディスカバリは行わない．これは，アプライアンスはあくまでもサービスを提供するだけで，アプライアンスの管理は全て PAC によって行われるためである．SOAP インタフェースは，Apache Axis によって実装されており，PAC からアプライアンスを制御する際のアクセスポイントとなる．Apache Axis はサーブレットとして Tomcat[17] [4] 上に配置されており，受け取ったコマンドを解析した後，適切なデバイスドライバにそのコマンドを転送することによってアプライアンスの操作をする．

また，Smart Appliances の提供する記述ファイルには RDF と WSDL の 2 つがある．RDF には，UPnP の記述ファイルを元にしたアプライアンスの機能やベンダ情報，また，アクセスポイントなどの論理的な情報など，アプライアンスを理解するた

[4] Jakarta プロジェクトのサブプロジェクトとして開発されているオープンソースのソフトウェアで、Java サーブレット・JSP を処理するアプリケーションサーバ

めのメタ情報が含まれる．一方で，WSDL ファイルは，SOAP の動的なリモートメソッド呼び出しを実現するためのデータタイプなどの，XML のメッセージを Java のメソッド呼び出しに変換するための情報が含まれる．本実装では，Apache Axis エンジンによって自動生成される WSDL ファイルを使用している．WSDL の例を付録 A.1 に示す．

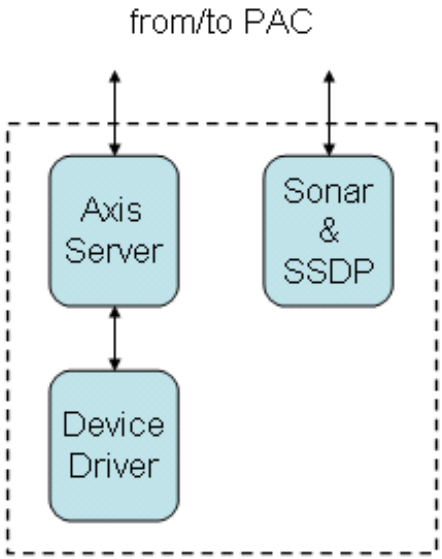

図 2.4: スマートアプライアンスの構造

RDF によるデバイス特性記述

RDF はセマンティック WEB 用に開発された XML を拡張した記述方式である．基本的な概念としては，主語（Subject），述語（Predicate），目的語（Object）の 3 つにより表現される[5]．またこの 3 つの組み合わせのことをステートメント（Statement），もしくはトリプルと表現し，複数のステートメントをまとめたものをモデルとする．

SENCHA におけるデバイスの内部情報の特性記述にはこの RDF を参考にしている．特に重要な長所のひとつは，プロパティに対しても一意に定まる URI（もしくは qaulified URI）を割り当てられることである．例えば，異なるベンダが同じプロパティ名を設定しても，URI によって識別することが可能である．また，検索に用途を限定した場合には，グラフ構造を完全に再現する必要はなく，三者関係を扱うことによって行えるという利点も備えている．

記述のフォーマットには，N-Trpiple[21] や Notation3[20] のような記述形式もあるが，これらは主に計算機による処理を念頭においてあるため，人間による可読性が低く，関係の連鎖を把握しずらいという欠点がある．そこで，SENCHA システムでは，XML による記述を採用している．

以下の図 2.5 に記述例を示す．この例では，スキーマの宣言の部分を省略している．重要な情報としては，RDF の論理的なロケーションを含む SSDP のメッセージング

[5]別の表現として，Resource，Property，Literal の組み合わせが用いられる場合もある

に必要な情報（ssdp タグ），コントロールのために必要な SOAP 関連の情報（soap タブ），そして，このデバイスの持っている機能の情報である（core:function タブ）.

```
<!-- subject -->
<rdf:Description rdf:about="urn:sencha:Light1">
  <!-- predicates and objects -->
  <ssdp:serviceName>urn:sencha:Light1</ssdp:serviceName>
  <ssdp:serviceType>upnp:sencha:Light</ssdp:serviceType>
  <ssdp:location>:8080/axis/property.rdf</ssdp:location>
  <ssdp:cacheControl>300</ssdp:cacheControl>
  <core:friendlyName>Light</core:friendlyName>
  <core:room>roomA</core:room>
  <core:owner>all</core:owner>

  <core:function rdf:resource="urn:homecomp:function:LightFunction" />
  <core:functionType>Light</core:functionType>

  <core:URLBase>http://localhost</core:URLBase>
  <core:WEBPort>8080</core:WEBPort>
  <soap:controlURL>/axis/services/LightService</soap:controlURL>
  <soap:interfaceWSDL>http://localhost:8080/axis/services/LightService?wsdl</soap:interfaceWSDL>

</rdf:Description>

<rdf:Description rdf:about="urn:sencha:function:LightFunction">
  <function:name rdf:resource="urn:sencha:function:LightFunction#power"/>
</rdf:Description>

<rdf:Description rdf:about="urn:sencha:function:LightFunction#power">
  <function:value>on | off</function:value>
  <function:mean>turn power</function:mean>
  <function:method>power</function:method>
</rdf:Description>
```

図 2.5: アプライアンスの特性記述の例：ライト

2.1.3 SOAP の考察

2.1.2 節にて述べたように，SENCHA ではアプライアンスの操作のための機構として SOAP-RPC を採用している．この節では，SOAP による制限を述べる．

SOAP について

SOAP[18] は，CORBA[6][22] や DCOM [7][23] のようなプロプリエタリなプロトコルを使用した RPC（Remote Procedure Call）の相互運用性を改善するために提案された仕様である．オブジェクトを XML という汎用的な文書に直列化することによって，ノード間の依存関係を減らし，疎結合を目指した RPC である．

また，RPC はあたかもローカルでのメソッド呼び出しのように，リモートのメソッド呼び出しを実現するためのメカニズムであり，ローカルには，リモートノードの代

[6]Common Object Request Broker Architecture．OMG によって策定されている分散オブジェクトの仕様.

[7]Distributed COM．Microsoft 社の提唱している COM（Component Object Model）を分散環境に適応した分散オブジェクトの仕様.

理となるプロキシが設置され，リモート側にはプロキシからのメッセージを処理するためのスタブが必要となる．つまり，クライアントはリモートメソッドのインタフェースに対して事前に合意を持つ必要があり，スタブを持っている必要があるということである．SOAP では，この依存関係を解決するために WSDL を利用し，クライアントがスタブを自動生成する余地を与えている．これによって，インタフェースの依存を動的に解決している．

データタイプの制限

SOAP における RPC は，SOAP エンベロープという XML 記述によってなされるが，扱うデータは XML スキームによって定められているプリミティブなデータタイプのみである．オブジェクトも扱うことはできるが，複雑に階層化されているもの，多次元配列などは扱えない．また，レスポンスに含まれる内部のオブジェクトを操作することができないという欠点もある．

セキュリティ

SOAP は，HTTP や SMTP のような転送プロトコルの上層に位置するプロトコルである．そのため，HTTP で実現しているセキュリティ機構とは別の機構を，SOAP のレイヤでも構築する必要がある[8]．このような機能の追加により，SOAP は重いプロトコルとなり，リソースに制限がある環境では不向きである．また，階層を増やすことによるシステムの脆弱性も増えるという部分も考慮しなくてはならない．

重い処理による制限

SOAP を用いた SENCHA では，デバイス間通信が PAC とスマートアプライアンスとの間に限定されているが，ユーザの要求するタスクを成し遂げるためのアプリケーションは，必ずアプライアンス間通信も必要になってくる．そのためには，WSDL の解析，SOAP メッセージの生成に付随する XML ファイルの解析というオーバヘッドの多い処理をアプライアンスに強制しなくてはならない．特に，PAC では，比較的リソースの豊富な PDA や小型のノートパソコンなどを期待できる一方で，アプライアンスはリソースがまだまだ乏しいと考えられるためそのような重い処理は不向きである．

バージョンによる互換性

SOAP は仕様であり，実装は各ベンダに委ねられている．そのため，過去にベンダ間の解釈の違いにより，相互運用性を得られなかったという問題があった[9]．また，SOAP のバージョンに関しても問題がある．機能を満載した仕様である SOAP であるが，バージョン 1.1 と 1.2 の間では，いくつかの大きな変更がある．その中のひとつが，データモデルの扱いであるが，1.1 では，独自のデータモデルを定義していたの

[8] SOAP エンベロープによるデータ転送により，HTTP の提供する機能とは別のセキュリティ対策が必要である

[9] コンフォーマンステスト（適合性試験）を用意することによって改善されつつある

に対し，1.2 ではより汎用性を持たせるために XML スキームの利用を指定している．また，このような変更に対して，1.2 の仕様では下位互換性を保つことを仕様では定めておらず，実装者の任意によるものとなっている．つまり，完全な互換性を得られない可能性を示唆していることになる．

ホームコンピューティング環境では，軽量なシステムであることが強く要求される．また，ソフトウェアのバージョン管理が難しい環境であるため，システムの基盤となる部分の整合性がバージョンによって左右されるのは好ましくない．このように SOAP はホームコンピューティング環境で不向きだと判断できる点もある．

2.2 REST: Representational State Transfer

REST[6][7][8] は，Fielding [10]によって提唱された，WEB アーキテクチャのモデルである．WWW が広く一般に普及した後，高いスケーラビリティ，相互運用性などの成功を成し得た基盤となる部分をソフトウェアアーキテクチャの観点からまとめたものである．Fielding は，このモデルのことを，アーキテクチャ的スタイル（Architectural Style）と呼んでいる．

REST 自体は，具体的な仕様を定めているわけではなく，アーキテクチャを構成する要素の役割と機能の定義と，それを制限する制約のセットと，それらの要素間の関連をまとめた WEB アプリケーションを構築する際のガイドラインの役割を担っている．REST はあくまでもモデルであるので，WEB は REST のひとつの実装例ということになる．

次節では，REST を説明する際に重要な概念となる用語の説明をする．続く，2.2.2 節で REST を構成する基本要素であるコンポーネント，コネクタ，データの 3 つを説明する．

2.2.1 REST の用語説明

REST における重要な概念であるリソースとリプレゼンテーションの説明をする．

リソース（Resource）

REST におけるリソースとは，情報に関する抽象化であり，名前を付けることのできるいかなる情報もリソースとして扱うことが可能である．例えば文書であるとか，一時的なサービス（今日の東京の天気など），複数のリソースを集めたもの，物理的なもの（実際の人，もの）などの情報をリソースとして扱うことができる．

リプレゼンテーション（Representation）

リプレゼンテーションとは，リソースの現在の状態，もしくは意図した状態を表現するものである．リプレゼンテーションは，HTML などの文書ファイル，JPEG などの画像ファイル，その他の音声ファイルや動画ファイルなどを用いて表現される．

[10]Roy T. Fielding．HTTP, URI の策定者の一人であり，Apache Software Foundation の創設者のひとり

2.2.2 REST の基本要素

REST のアーキテクチャを定義するために，コンポーネント，コネクタ，データの三つの基本要素が定義している．

- コンポーネント[11]
  REST で定めているコンポーネントとは，ネットワーク上の各ノードのことでオリジンサーバ（origin sever），ゲートウェイ，プロキシ，ユーザエージェントの４つのタイプがある．

  オリジンサーバは，リソースに対するリプレゼンテーションのソースとなるものであり，ユーザエージェントはいわゆるブラウザなどの WEB クライアントのことを指す．プロキシとゲートウェイは，オリジンサーバ，ユーザエージェントを仲介するコンポーネントであり，その違いはクライアントがコントロールできるか否かという点にある．ネットワーク，もしくはオリジンサーバによって，仲介することを設定されるのがゲートウェイであり，ユーザによって仲介することを設定されるのがプロキシである．仲介コンポーネントの目的は，他サービスのインタフェースのカプセル化，データ変換，パフォーマンス向上，セキュリティ強化などである．

- コネクタ
  コネクタは，各コンポーネントが保持している抽象的なインタフェースのことである．各コンポーネントは，このコネクタを通じて他コンポーネントと通信する．コネクタのタイプとしては，クライアントコネクタ，サーバコネクタ，キャッシュコネクタ，リゾルバコネクタ，トンネルコネクタがある．

  REST における通信はすべてステートレス（Stateless）である．つまり，上記のコネクタ間で転送されるリクエスト内に，コネクタがリクエストを理解するために必要な情報をすべて含んでいるということである．例えば，今受け取ったリクエストが，以前に処理したリクエストと依存関係があるというような情報も含めることが可能である．

  これによって，コネクタがリクエストを処理するにあたってアプリケーションの状態を保持する必要がなくなり，物理リソースの節約とスケーラビリティを実現している．また，コネクタがインタラクションの意味までを解釈せずに，インタラクションを並列に処理することができる．さらに，リクエスト内にキャッシュ可能などうかを判断するための情報を含ませることを強制することによるスケーラビリティの向上などの特徴を導き出している．

- データ
  REST におけるデータで重要な概念は，2.2.1 節で述べたリソースとリプレゼンテーションの２つと，リソースを一意に識別するためのリソース識別子である．これらのリソース，リプレゼンテーション，リソース識別子によって構成されるデータを分散オブジェクトにおけるデータの概念と比較すると，分散オブジェクトではすべてのデータがオブジェクト内にカプセル化されているのに

[11] ここで使用しているコンポーネントという用語は，2.1.1 節で述べた OSGi におけるコンポーネントとは異なる．2.1.1 節のコンポーネントは，ソフトウェアを管理する際のモジュールを指し，本節のコンポーネントは，ネットワークを構成するノードを指す．

対し，REST では，すべてのデータが外部からアクセス可能であるという点が一番の違いである．

REST では，データタイプに関するメタ情報を用い，表 2.1 に示される 3 つの方法を取り混ぜる形でデータ転送を行うことを定義している．

	方法	長所	短所
1	サーバにてデータ処理を行い，固定された形式でクライアントに送信する	データの詳細をクライアントから隠蔽することができ，クライアントの実装を簡単にすることができる．	サーバの負担が大きいためスケーラビリティが損なわれる．
2	データ処理エンジン内にデータをカプセル化し，クライアントに送信する．	データをデータ処理エンジン内に隠蔽できる同時に，処理機能を提供する．	ユーザにデータ処理エンジンの選択肢がなく，ネットワークトラフィックが大きくなる．
3	処理されていないバイナリデータを，メタ情報と共にクライアントに送信する．	ネットワークで転送するデータの量を最小限にし，サーバの負担を減らすことによりスケーラビリティを得ることができる．また，クライアントは自由にデータ処理エンジンを選択することができる．	情報の隠蔽をすることができず，また，サーバ，クライアントでデータに対する共通の理解が必要になる．

表 2.1: データの転送方法

REST では，上記の 3 つの基本要素を定義し，制約を設けることによって，WEB アプリケーションの振る舞いを決定している．

2.2.3 基本原理

REST の基本となる原理は，以下の項目から成り立っている．

リソースの一意な識別

2.2.1 節にて述べたように，REST では抽象化された情報をリソースとして扱っている．WWW では，URI（Uniform Resource Identifier）[24] による統一されたアドレス空間を実現しており，個々のリソースに URI を指定することによって，リソースの一意な識別が可能である．

リプレゼンテーション（Representation）を用いたリソースの操作

REST では，リソースの状態をリプレゼンテーションを用いて表現すると定義している（2.2.1 節参照）．このリプレゼンテーションをコンポーネント間で転送することによって，リソースの状態を変更することができるものと定義している．

自己記述型メッセージ（Self-descriptive messages）

REST では，メッセージの受け取り先が受信したリクエストを理解するための情報をすべてメッセージ内に記述している．例えば，リソース識別子，データフォーマットやコントロールデータなどが含まれる．このように必要な情報がメッセージ内に記述されることによって，サーバ側でクライアントの状態を管理する必要がなくステートレスな接続を実現している．

ハイパーテキストによる状態マシン（State machine）

基本原理の具体的な流れは図 2.6 に示す．REST におけるリソースは WWW では WEB サーバのことを指す．リソースの状態はリプレゼンテーションという形で表され，リプレゼンテーション内に含まれるハイパーリンクによって，WEB アプリケーションの状態マシンを形成している．図 2.6 の左の図では，クライアントがリソースに対してリクエストを送り，リプレゼンテーションを取得した状態を示している．このようにクライアントがあるリプレゼンテーションを保持している状態が，WEB アプリケーションの状態を表していることになる．この状態で，ユーザがリプレゼンテーション内のリンクをクリックし，別の URI に対してリクエストを送信すると，その URI の指し示すリプレゼンテーションテーションを新たに受け取り，WEB アプリケーションも別の状態へと遷移することになる．このようにリソースを表す状態（Representational State）を転送（Transfer）することが REST の基本原理である．

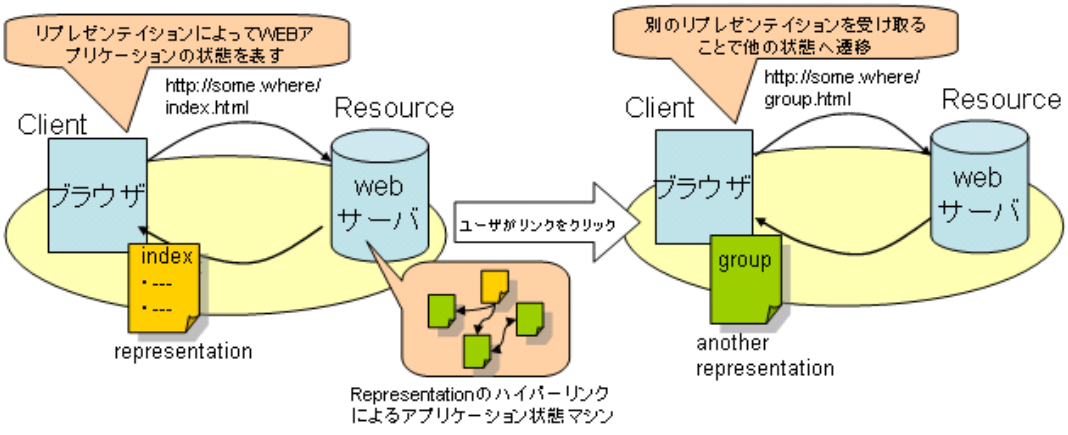

図 2.6: Representational State Transfer の原理

2.2.4 REST のまとめ

この節では REST の特徴を述べた.

- コンポーネント間相互関係のスケーラビリティ
- インタフェースの汎用性
- コンポーネントの独立した配置・展開（deployment）

WWW が成功した理由は，この 3 つの要件を満たすべくアーキテクチャが設計されているためである．まず，一つ目のスケーラビリティは，ステートレスな通信によるサーバの負担の削減，キャッシュ，または，キャッシュプロキシによるネットワークトラフィックの削減によって実現している．二つ目のインタフェースの汎用性は，HTTP という固定された意味を持つ少数セットのコマンドを用いることによって実現している．三つ目のコンポーネントの独立した配置展開に関しては，汎用的なインタフェースの実現と，明確な各コンポーネントの振る舞いの定義があるために実現している．

第3章　設計・実装

前章で説明した通り，REST はネットワークアプリケーションを構築する際のガイドラインとなるモデルである．特にデータをリソースとして表現し，固定された意味を持つ少ないコマンドによってそのリソースを操作するということをモデル化している．このモデルに従えば，スケーラビリティ，拡張性，または，独立したコンポーネントの展開などが可能になるというものであった．

このモデルを SENCHA に適用することによって，簡潔にスマートアプライアンスを実装することが可能になり，簡潔なコマンドによるコミュニケーションを利用することにより，汎用的なインターフェースを実現できると考えられる．これによって，高い相互運用性と，エンドユーザによる自由なアプリケーションの構築を実現する．

シンプルなコンポーネントとしてのデバイスの実装，シンプルなコマンドによるコミュニケーションによる汎用的なインタフェースによって，高い相互運用性と，エンドユーザによる自由なアプリケーションの構築が可能になると考えられる．

3.1　SENCHA の再分析

この章では，SENCHA を REST の視点から分析する．

UPnP

まず，SENCHA の根幹に位置する UPnP に着目すると，UPnP 自体が REST に近いアーキテクチャであることがわかる．一つ目には，SSDP，GENA などの HTTP を拡張したプロトコルを利用している点が挙げられる．特に，デバイス発見，イベント通知のために必要な最小限のコマンドを，拡張 HTTP コマンドとして実装しているため，二者間のメソッド呼び出しをシンプルなものとして実現できている．二つ目には，UPnP デバイスの内部表現が，デバイス，サービス[1]共に URI によって一意に識別できる点が REST のモデルに則している．しかし，サービスに関しては，SOAP インタフェースを経由するため，サービスの内部に存在するデータに対しては URI での参照はできないという部分が REST に則していない点といえる．

続いて，2.2.2 節で説明した REST における 3 つの基本要素の観点から分析を行う．

コンポーネント

SENCHA の対象としている環境は，現状では部屋や家などのひとつのローカルネットワークによって構成される閉じた空間である．そのため，外部とのコネクションと

[1] ここで使用しているサービスという用語は，2.1.1 節で述べた OSGi のサービスとは異なり，UPnP デバイスが提供する「機能」のことをサービスという言葉を用いて表現する．

なるようなプロキシ，ゲートウェイなどのコンポーネントは必要としない．

SENCHA の想定する環境は，2.1.2 節で述べたように，PAC とスマートアプライアンスによって構成される．これらを REST の用語を用いて分類すると，PAC はクライアント，スマートアプライアンスは大まかに言ってサーバとしての関係になる．しかし，スマートアプライアンス同士の関係を考察すると，対等の Peer-to-Peer の関係であるため，スマートアプライアンスはクライアントとしての意味も持つことになる．

厳密に述べるならば，現状で実装している X10 モジュール [19] を利用したライトアプライアンスは，シリアルケーブルで接続した X10 モジュールを，パーソナルコンピュータ（PC）を用いて制御しているという意味では PC がゲートウェイと解釈できる．しかし，本論文ではスマートアプライアンスとして扱っているアプライアンスは，物理的なデバイスではなく，論理的に表現されているアプライアンスなため，特に，PC によるゲートウェイによって実装されているか否かの差異は扱わないものとする．

また，将来的に，デバイス操作の占有・共有や，競合解決を考慮する場合，PAC が他 PAC のプロキシとなる可能性も考えられる．

コネクタ

REST におけるコネクタは以下のような種類のものがあった．

- クライアントコネクタ
- サーバコネクタ
- キャッシュコネクタ
- リゾルバコネクタ
- トンネルコネクタ

まず，WWW の環境と SENCHA の環境との大きな違いを挙げると，WWW はドキュメントなどの静的なデータを主に取り扱うため，HTTP GET コマンドが主な操作であるのに対し，SENCHA ではスマートアプライアンスをコントロールすることを目的としているため，HTTP POST によるコントロールが主であり，かつ，それによりリソースの Representational State を表すデータは動的なものになるためにキャッシュは使えないという点がある．唯一，静的なデータであるアプライアンスの特性記述ファイル（RDF）に関しては，WWW のキャッシュの方法とは異なるが RDFDB に保持されるため，SSDP のコネクタはキャッシュコネクタと考えることができる．また、SSDP のコネクタは，サーチ，アドバタイズといった標準の HTTP には概念も追加されているので，サーチコネクタ，アドバタイズコネクタが追加される．

また，前節にも述べた通り SENCHA の想定している環境ではローカルな閉じた環境を想定しているため，リゾルバコネクタやトンネルコネクタも必要ないことが分かる．

データ

REST におけるデータの観点から述べると，SENCHA のデータ表現方法は，REST のそれとほぼ同じである．スマートアプライアンス URI によって識別できるデバイ

ス，サービスが存在し，それらを表すリプレゼンテイションがある．しかし，サービスに関するリプレゼンテーションは，SOAP エンベロープによって表現されており，その内部に URI では一意に識別できないデータ構造を持つ点が異なる．

3.2 REST を用いた基本設計

接続の種類（コマンドの種類）とそれに対するリソースの分類を行う．

3.2.1 リソースの種類

SENCHA におけるリソースの種類は，大きく分けて以下の 3 つに分類できる．

- デバイス
  デバイスリソースは，デバイス全体を指すリソースで，リプレゼンテーションは静的な RDF ファイルである．RDF ファイル内にはデバイスのベンダや機能，IP アドレスやポート番号の論理的な情報などのデバイスに関するメタ情報と，そのデバイスが持っている機能リソースや，データシンク/ソースリソースへの参照と，それに対するメタ情報を含んでいる．デバイスリソースのサブリソースとなる機能リソースのメタ情報内には，予め機能リソースの取りうる状態が定義されている．このデバイスリソースは，静的なリプレゼンテーションしか持ち合わせないため，POST によるコントロールは受け付けず，GET によるリプレゼンテーションの取得のみを許可する．

  クライアントとなる PAC はこのデバイスリソースを取得し，解析することによって，デバイスの持っている機能を把握することが可能となる．

- 機能
  機能リソースは，そのデバイスの持っている最もプリミティブなリソースとなり，power や channel といった最小単位の機能を表している．最小単位の機能を表しているため，機能リソースの取り得る状態は簡単な状態遷移図として表すことが可能である．この情報はデバイスリソースのリプレゼンテーション内（RDF）に記述されているが，これによって，クライアントとなる PAC からの POST による状態変更を容易に行える．状態変更は，遷移後の状態を表す単純な text/plain の文字列をボディとして HTTP POST によって転送される．

  また，機能リソースには，操作可能 (controllable) なものと，イベント発行が可能 (monitorable) なものに分類される．これらの属性も RDF ファイル内に記述され，controllable な機能リソースは POST による状態変更を受け付けることができるリソースのことを指す．monitorable なリソースは状態変更を監視することが可能なリソースで，後述する HTTP SUBSCRIBE を受け付けることのできるリソースである．例えば，センサを組み込まれたスマートオブジェクトが，外部からのコントロールを受け付けず，リソースの状態が変更したというイベントを発行する機能リソースを持っていた場合，そのリソースには，monitorable の属性が付加され，controllable の属性は付加されない．これらの情報も前述のデバイスリソースである RDF ファイル内に記載される．

- サービス
  サービスリソースは，機能リソースとロジックを組み合わせており，単純な状態遷移図では表現できないリソースと定義する．そのため，予め取りうる状態を把握できないので monitorable な属性は付加することができず，POST によるコントロールのみ可能である．POST する際の引数（HTTP ボディ内に含まれる）は複数定義することも可能である．HTTP ヘッダ内の Content-Type は，appliacation/x-www-urlencoded を使用し，複数の「属性＝値」の組をカンマ「,」で区切った形で表現される（3.2.3 節参照）．

  具体的なサービスリソースの例として，TV の「Channel Search サービス」を挙げる．このサービスリソースは，Channnel（機能リソース）のラッパとなっており，チャンネルを順にサーチしていくサービスを提供する．サーチする際の切り替えるタイミングを管理する「インターバル」というロジックを持っており，このインターバルに秒数をセットすることによって，タイミングを設定する．デフォルトでは，全チャンネルに対してサーチを行うが，サーチするチャンネルのセットを設定するための「チャンネルセット」というロジックも含む．

  このようなサービスを想定する場合，POST のボディには以下のような内容を含むことになる．

  ```
  interval=5,channel_set=4&8&10
  ```

  この例が示すのは，４，８，１０の３つのチャンネルをインターバル５（単位は TV の定義による）でサーチしていくという意味を表す．

- データシンク/データソース
  データシンクやデータソースは，メディアデータを扱うリソースで，データシンクはソースの保持しているデータへの URI を保持することができ，データソースは，複数のメディアデータを公開しており，データソースを指す URI のクエリ（例：`http://some.where/data_source?foobar.mp3`）で各データを参照できる．データソースへの GET リクエストは，保持しているリストを返すか，もしくは，特定のデータを返し，データシンクへの GET は，現在設定されているソースの URI を返す．

  データソースの保持するメディアデータは，対応プロトコルと MIME タイプによって識別され，データシンクはプロトコルと MIME タイプに対応できる場合のみ，そのメディアデータを受信することができる．

  これらのプロトコル，MIME タイプと言うデータリソースに関するメタ情報もデバイスリソースの RDF ファイルに保持される．

3.2.2 通信の意味の分類

SENCHA 環境において，現時点で考えうるデバイス間接続の意味を分類すると以下の５つに分けられる．

- コントロール（HTTP POST）

- 状態取得（HTTP GET）
- イベント通知（HTTP SUBSCRIBE）
- アドバタイズ・サーチ (HTTP NOTIFY/M-SEARCH)
- メディアデータ転送（HTTP POST により接続先を設定．その後の通信はデータリソースに依存）

REST のモデル定義によると，必ずしも HTTP の標準のコマンドを使用するという制約を課していない．重要なことは，固定の意味を持った少数セットコマンドによってインタフェースを定義することである．つまり，SENCHA の想定しているホームコンピューティング環境で必要なコマンドは，POST，GET，SUBSCRIBE，NOTIFY，M-SEARCH の５つであると定義できる．

GET，POST に関しては，HTTP の仕様で定義されているもので，SUBSCRIBE は GENA(General Event Notification Architecutre)[25] によって定義されているコマンドである．NOTIFY，M-SEARCH は SSDP で定義されているコマンドである．

3.2.3 リプレゼンテーション

SENCHA 環境でやり取りされる通信の意味は，上記のような単純なものに限定されている．そのため，扱うデータも複雑なデータ構造を持つ必要はないと解釈できる．特に，リソースの状態を表現するためのリプレゼンテーションを表現する際に，階層構造を持つ構造体やオブジェクトである必要はなく，純粋に文字列の持つ意味が解釈できれば充分と判断できる．そこで，リプレゼンテーションの表記には単純な ASCII 文字列を利用する．同様の環境としては，データタイプを持たない変数や値を扱う UNIX シェル環境が挙げられる．UNIX シェル環境では，単純な操作の組み合わせによって処理が行われるため，変数や値がデータタイプを持たずとも破綻することなく必要な処理を行うことができている．

MIME タイプは，ASCII 文字列の表現としてを text/plain として扱い，複数の property=value のペアを扱う必要がある場合は，CGI などで利用されている application/x-www-urlencoded を使用することができる．また，メディアデータを扱う場合は，そのデータタイプを表現する MIME タイプを指定する．

3.2.4 全体像

REST 導入後の設計を以下の図 3.1 に示す．この図では各コネクタ間の関係を示している．

SENCHA におけるコネクタは，リソースと通信の分類から，クライアントコネクタ，サーバコネクタ，キャッシュコネクタ，サーチコネクタ，アドバタイズコネクタに分類できる．

PAC 側では，Appliance Controller コンポーネントと UI Constructor コンポーネントが，REST 対応のものへと変更されている．スマートアプライアンス側では，SOAP で実装されていたサービスがリソースとして各サーブレットで実装されるようになり，また，サブスクリプションのためのレジストリ（図中の SubReg=Subscription

Registry）と Publisher として HTTP POST を実行するコンポーネントが追加された点が変更点である．

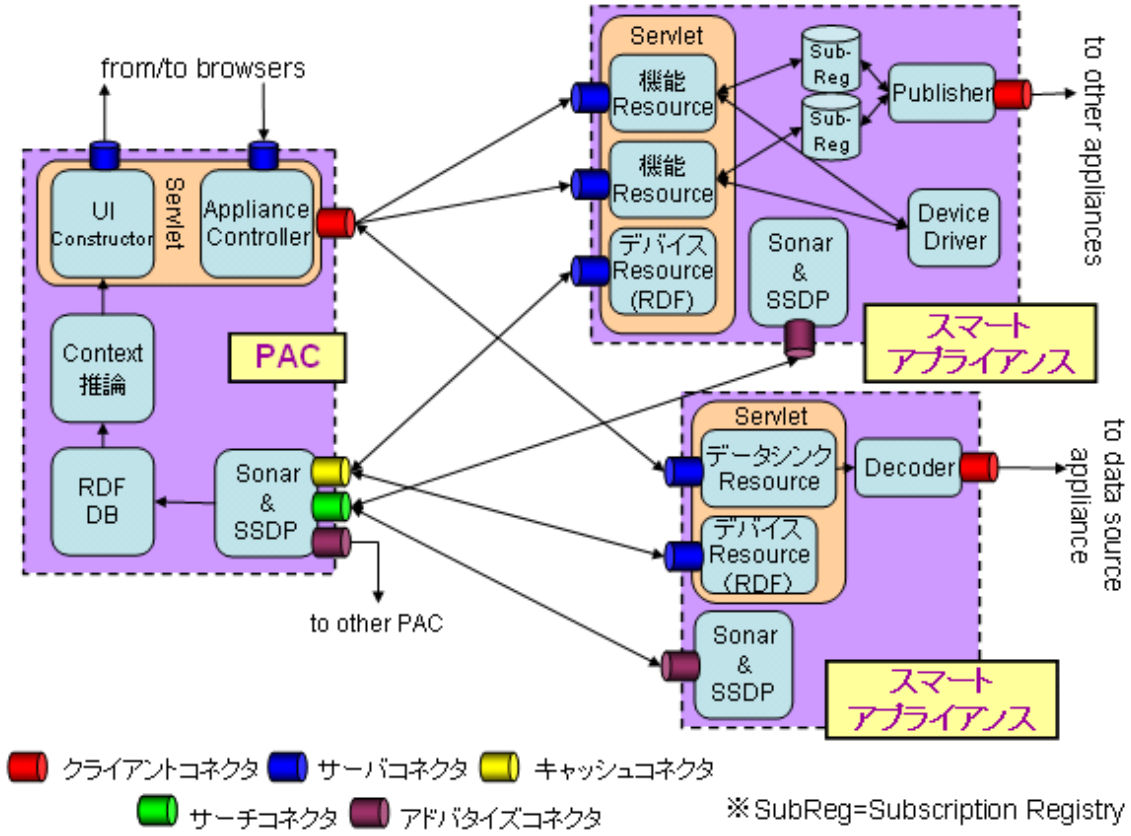

図 3.1: REST を導入した設計

[6] の REST との違いは，明確にクライアント/サーバの分離がなされているアーキテクチャではない点である．SENCHA の想定する環境では，スマートアプライアンスの役割は単にクライアントからのコマンドを処理するサーバの役目だけではなく，他スマートアプライアンスのためのクライアントにもなりうる，Peer-to-Peer な環境にある．そのため，サーバ側にもクライアントコネクタが存在する．

3.3 使用例

この節では，REST を用いた場合の SENCHA の動作フローを示す．

コントロール

図 3.2 では，初期動作からコントロールまでを示す．

まず，初期動作としては，PAC が環境内に存在するアプライアンスの情報を取得しなくてはならない．アプライアンス情報の取得には，2.1.2 節で説明したように SSDP が用いられ，PAC が新たにネットワークに接続した場合はサーチが行われ，環境に新

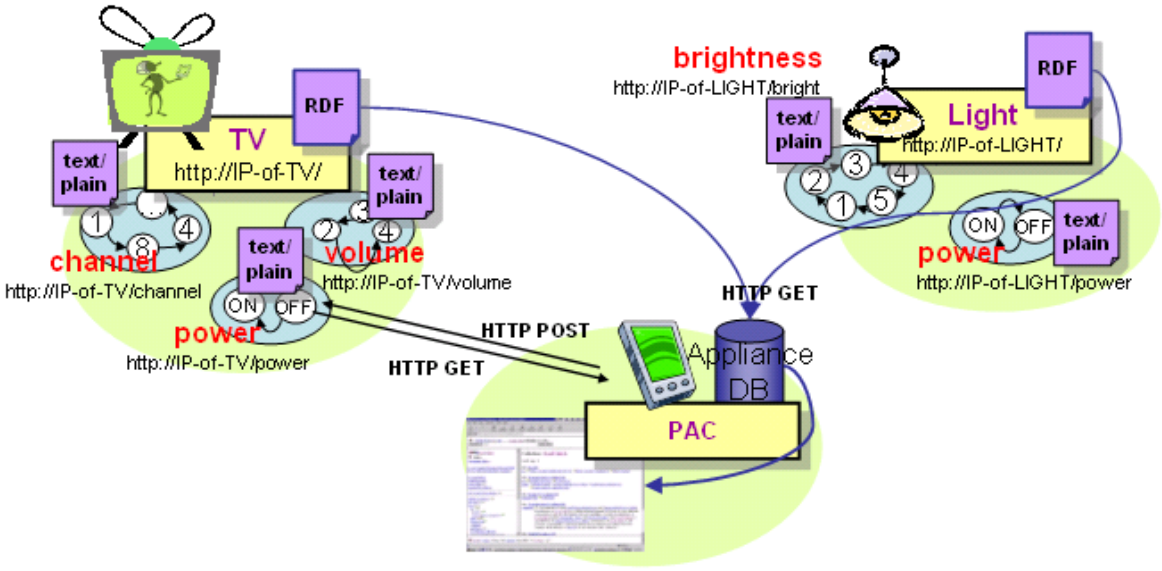

図 3.2: 使用例：コントロール

たなアプライアンスが追加された場合は，アプライアンスが自動的にアドバタイズすることによって，PAC にアプライアンス情報が供給される．

PAC は取得した情報を下にユーザインタフェースを生成する．現状のアプローチでは，HTML を生成しているが，HTTP と URL が扱えるシステムであるならば，他の方法によって生成することも考えられる．例えば，Macromedia Flash や，Microsoft の PowerPoint などが利用できる．

ユーザインタフェースは，機能ごとに用意され，コントロール（図 3.3 参照），イベント接続（図 3.4 参照），データ接続用（図 3.7 参照）の UI を生成している．

これによって，PAC はその環境における必要な準備を終わらせたことになる．ユーザは，コントロール UI を操作することによって，アプライアンスをコントロールすることができる．実際には，HTML からのメッセージは，PAC 上で適切な POST メッセージに変換され，アプライアンス上のターゲットとなる機能リソースやサービスリソースに転送される．

イベント通知接続

イベント通知接続は，イベント接続 UI（図 3.4 参照）によって操作される．イベントの発生源となるデバイスの機能リソースと，その状態を設定することによって，その状態になった時にイベントを発行することができる．イベントの通知先は必ずしも PAC である必要はなく，他アプライアンスへ直接イベントを通知することが可能である．この時，イベント発生源の状態が変化したという内容を通知するのではなく，イベント通知先の機能リソースの状態を変更するように設計している（図 3.5 参照）．これによって，PAC を仲介させず，直接アプライアンス間での関連付けが可能になる．図 3.5 の例では，TV のボリュームが 4 か 5 に設定された場合，ライトの電源をオンにするという関連付けをしている例である．

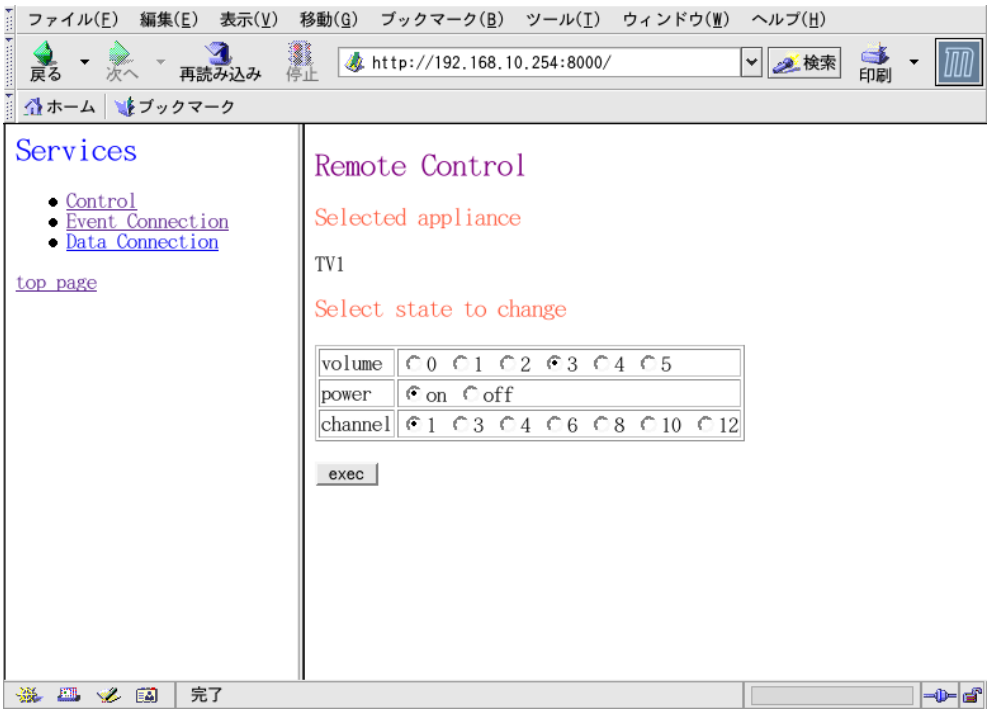

図 3.3: コントロール UI

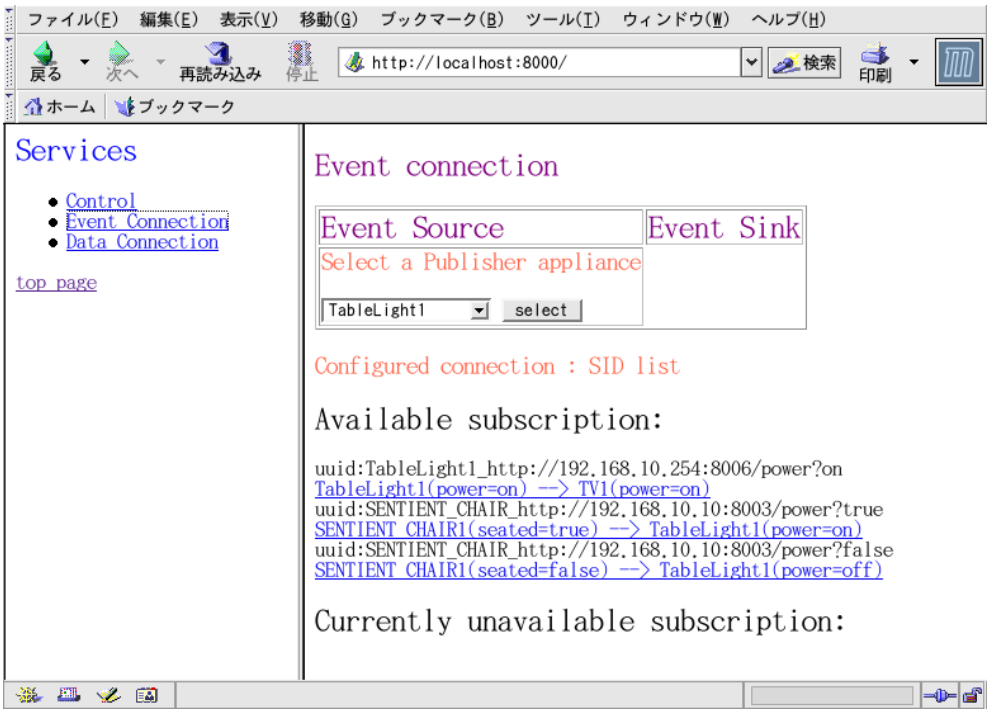

図 3.4: イベント通知接続 UI

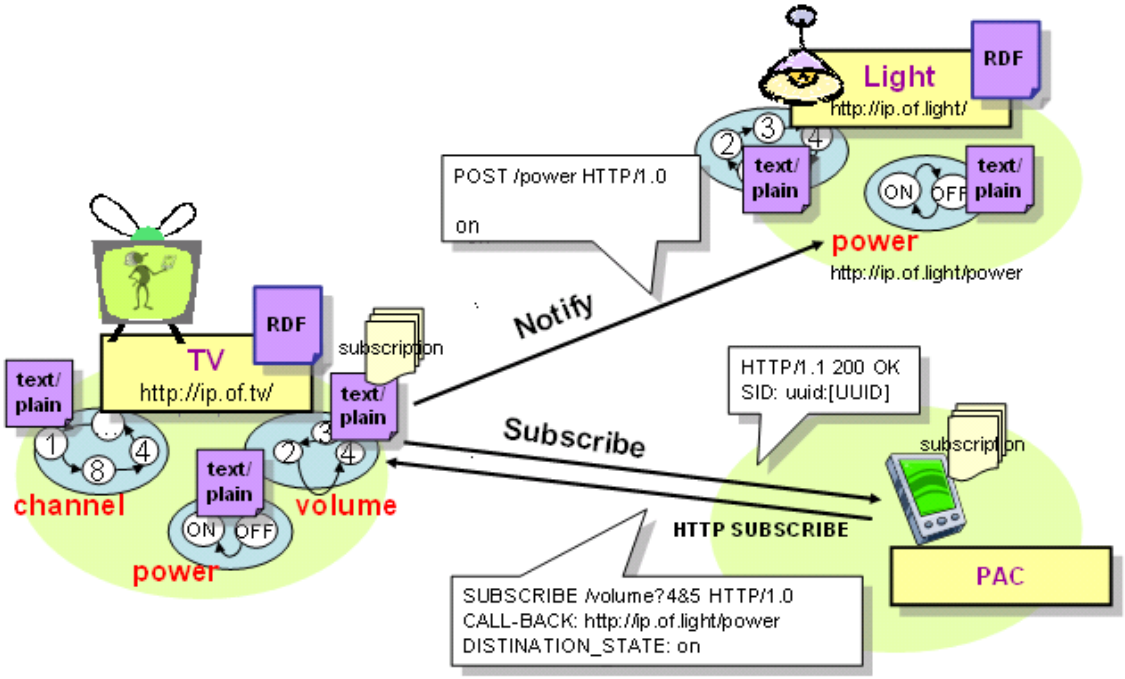

図 3.5: 使用例：イベント通知接続

また，このサブスクリプション（イベント購読）の情報は，PAC とイベント発生源となるアプライアンスの両者によって管理される．これによって，イベント通知先のアプライアンスは，サブスクリプションに関しての情報を保持する必要がなく，イベント発生源のアプライアンスからの通知によるコントロールなのか，PAC からユーザが行っているコントロールなのかの区別をする必要がないというステートレスな接続になっている．

サブスクリプションには SID[2] が発行され，これにより，この SID を所有している PAC からのみイベント通知接続の削除が可能である．また，この SID は，イベント発生源と通知先を一意に識別できる文字列で表現しているため，同じ組み合わせのイベント通知接続は二重に設定することができないようになっている．

さらに，イベント発生源となるアプライアンスがサブスクリプション情報を管理するため，イベント通知接続の状態を永続させることもできる．初期設定では，サブスクリプションを設定した PAC がその環境から去ると，そのサブスクリプションは無効になるが，図 3.4 中に示す SID をクリックすることによって永続するように設定可能である．初期設定の状態だと，PAC が再びその環境内に入ると自動的にサブスクリプションが再設定される．

データ転送

転送されるメディアデータは，MIME タイプと URI のスキームによって定義される．データの転送に関わるアプライアンスは，データシンクとなるリソースを持っているか，もしくはデータソースとなるリソースを持っているアプライアンスである．

[2]Subscription ID

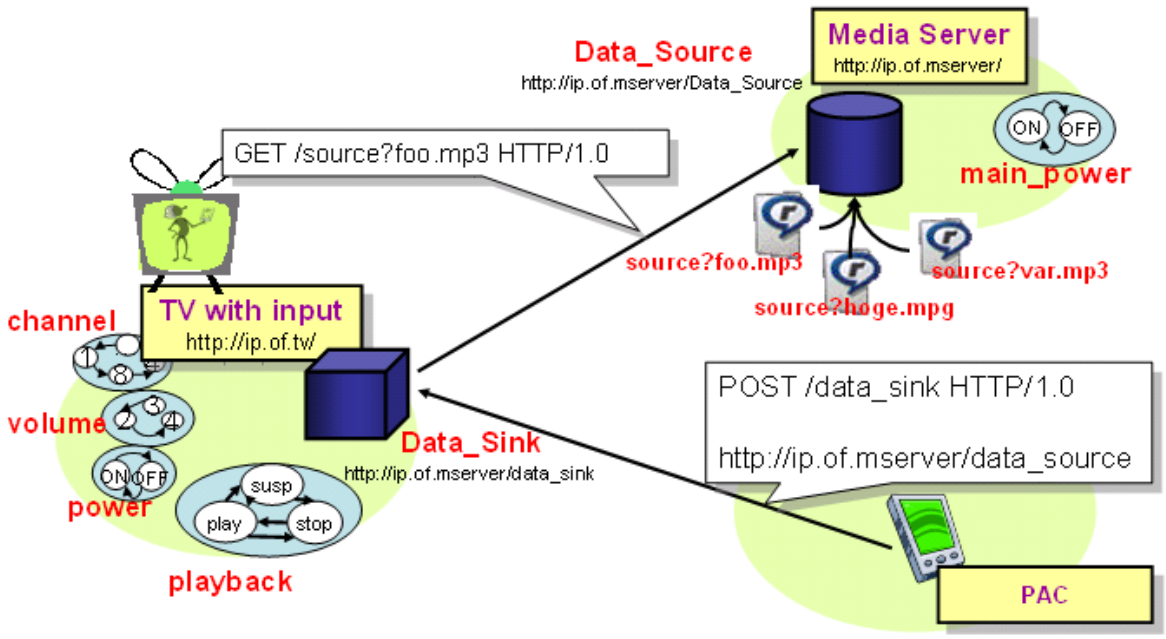

図 3.6: 使用例：データ転送

図3.6では，PACからTV[3]のData_Sinkリソースに対して，Media ServerのData_Sourceとなるリソースの URL を POST している（POST のボディに Data_Source の URI を指定）．これによって TV の Data_Sink にメディアデータの URI がセットされ，データをプル（pull）することができる．PAC からの POST のボディに指定される URI のスキームは必ずしも http ある必要はなく，Data_Source が別のプロトコルをサポートしているならば，そのプロトコルをスキームに指定することも可能である．Data_Sink は指定されたプロトコルにて，Data_Source からデータを取得することができる．この後，TV のコントロールにより，Playback リソースを play の状態に変更することによって，動画が再生されるという例である．コントロールは図 3.7 に示す UI で行う．図 3.7 は，TV のコントロール画面を表しており，処理できるプロトコルは HTTP で, データタイプは video/mpeg，audio/mpeg である．中央のデータを選択する部分では，プロトコルとデータタイプの条件にあてはまるメディアデータのリストが表示される．このデータは，同じネットワークセグメント内にある Media Server から提供されている．

3.4 Java による仮想アプライアンス

今回の実装では，いくつかの仮想的なスマートアプライアンスを Java によって構築している（表 3.1 参照）．

表中には示していないが，各々特性記述（RDF ファイル）を保持するデバイスリソース（3.2.1 参照）を持っており，HTTP GET による状態の取得はどのリソースも

[3]JMF による動画再生アプリケーション

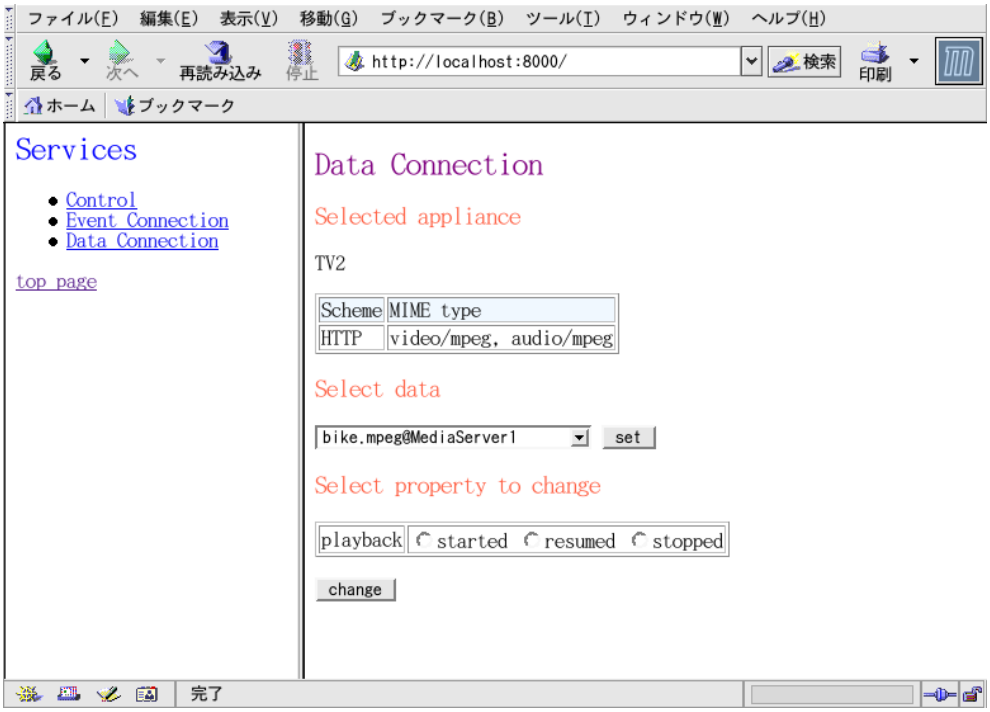

図 3.7: データ転送 UI

可能である．monitorable 属性を持っているリソースは HTTP POST による状態変更を受け付ける[4]．

また，monitorable な属性を持っているリソースはイベントを発行が可能であり，controllable な属性を持っているリソースとイベント通知接続が可能である．イベント通知接続は，monitorable なリソースのある状態を指定し（複数の状態を指定することも可能），イベント通知先に controllable なリソースを指定する．このとき通知先のリソースの状態も指定することによって，イベントが起きるとその状態を POST することができる．

表中に示す 7 つのアプライアンスがある環境を想定すると，イベント通知接続は全部で 217 通りが考えれらる[5]．

[4]TV の Data_Sink リソースは，メディアデータの URI を POST により設定可能
[5]イベント発行元と通知先に同じリソースを設定できないと考える場合

アプライアンス	リソース	取り得る状態	controllable	monitorable	イベント通知先数
Table Light	power	on, off	○	○	14
	brightness	1, 2, 3, 4, 5, 6	○	○	24
Ceiling Light	power	on, off	○	○	14
TV	power	on, off	○	○	14
	channel	1, 3, 4, 6, 8, 10, 12	○	○	49
	volume	1, 2, 3, 4, 5	○	○	35
TV 外部入力付き	power	on, off	○	○	14
	playback	played, resumed, stoped	○	○	21
	Data_Sink	(メディアデータの URI)	(○)	×	
Media Server	Data_Source	(メディアデータ)	×	×	
Bed	isSomeBodyOn	true, false	×	○	16
Chair	seated	true, false	×	○	16

表 3.1: 仮想アプライアンス一覧

第4章　評価・議論

本研究では，ホームコンピューティング環境における相互運用性，拡張性という要件を満たすための手段としてRESTを導入を試みた．実際にホームコンピューティング環境においてRESTが有用であるかどうかを，定量的・定性的評価を用いて考察する．

4.1　定量的評価

この節では，実際にテストを行うことによる定量的評価を行う．

4.1.1　実験環境

実験環境には，２台のPCを用い，一方をJavaで実装された仮想アプライアンスを動作させるために使用し，もう一方をアプライアンスを操作するためのクライアントを動作させるために使用している．PC間の接続はクロスケーブルによって行い，tomcatはバージョン4.1.29，axisは1.2RC2を使用している．

	ホスト１（クライアント）	ホスト２（アプライアンス）
CPU	celeron 800MHz	Pentium III 866MHz
メモリ	384MB	384MB
OS	RedHat9	RedHat9
JavaVM	1.4.2_04	1.4.2_06
NIC	Intel(R) PRO/100 VE	Intel(R) PRO/100 VE

表 4.1: 実験環境

4.1.2　実行時間の比較

RESTを用いた実装と，SOAPを用いた実装における実行時間の計測を行う．SENCHAでは，SOAPを用いた実装はtomcatを用いているが，RESTはoscarフレームワークの提供するJettyベースのWEBサーバを用いている．しかし，今回のテストでは条件を同じにするため，両者ともtomcatを使用している．アプライアンスは，Javaによって仮想的に実装されたライトを用い，SOAP実装もREST実装も同じロジックを使用している．機能リソースはpowerのみを持っており，powerリソースの取りうる状態はonとoffの二種類がある．クライアントプログラムは事前にこの情報を知っていると仮定し，単純なHTTPメソッドを送信するプログラム

と，SOAP メソッドを呼び出すシンプルなクライアントによって行った．REST クライアントは java.net.HttpURLConnection クラスを使用し，SOAP クライアントは Apache Axis[16] によって提供されるライブラリを使用しており，XML のパーサには xercesImpl[29] を使用している．

テスト内容は，REST，SOAP，それぞれの通信方法で状態の取得，変更（アプライアンスの操作）にかかる通信時間の測定を行った．連続して送信するクエリの数を１，５，１０，１５，２０回と変えながらテストを行い，それぞれ 30 回行った結果を平均している．図 4.1 に結果を示す．

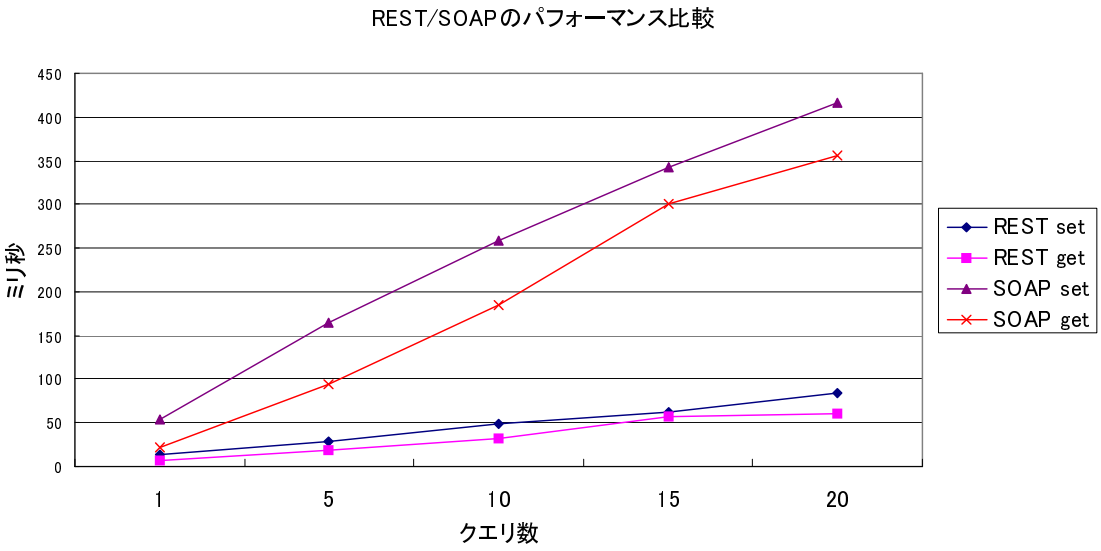

図 4.1: REST と SOAP による通信時間の比較

図に示すように，実行結果に明らかな差がでることが分かる．SOAP によるメソッド呼び出しでは，XML にエンコードされたデータをやりとりするため，その解析をするための時間がオーバーヘッドになって現れていると考えられる．

また，クラスのロード時間に相当な時間がかかることから（SOAP の場合 2000 ミリ秒以上），純粋な通信時間の測定をするため，クライアント起動後の初回の通信は測定していない．

4.1.3 メモリ容量の比較

4.1.2 節のテストでは，SOAP クライアントに Axis を利用した．Axis を使う際に必要なライブラリは以下のものがある．各 jar ファイルの容量は利用しているクラスの分のみである．

一方で，REST を用いた実装を行う場合，必要な外部ライブラリはアプライアンスの実装にサーブレットのライブラリ（表中の servlet.jar）が必要なだけである．SOAP による実装では，クライアント，アプライアンス，共に上記のライブラリが必要になる．つまり，SOAP に比べ REST を用いることによって，大きくメモリ容量を節約できるということが言える．

jar ファイル名	サイズ（kb）
axis.jar	1100
jaxrpc.jar	35
commons-logging.jar	31
commons-discovery.jar	66
saaj.jar	18
servlet.jar	78
合計	1148

表 4.2: SOAP：使用パッケージ容量

4.1.4 パケットサイズ

4.1.2 節のテスト行った際のパケットサイズを測定した[1]．表 4.3 に示す通り，SOAP を用いた場合の方がパケットサイズは大幅に大きくなっていることが分かる．これは REST では単純な文字列のみを HTTP メッセージボディのコンテンツとしているのに対して，SOAP ではリクエスト，レスポンス共に SOAP エンベロープを含んでいるためである．このように，ネットワークリソースの節約という観点から考察しても REST による利点は大きいことが言える．

	SOAP(bytes)	REST(bytes)
リクエスト	789	291
レスポンス	684	223

表 4.3: パケットサイズの比較

4.2 定性的評価

4.2.1 ユーザによる自由な接続

ホームコンピューティング環境では，1.2.1 節で示したエンドユーザプログラミングという要求がある．ユーザの自由な解釈により，道具やものの使い方をアレンジし，独自の方法で使用したいという要望に対応できることが必要である．

今回の REST 導入によるイベント通知接続の実現により，アプライアンス同士の関連付けを自由に行えるようになったことが，エンドユーザプログラミングに貢献していると考えられる．特に，UPnP で採用されているイベントシステムである GENA（Genaral Event Notification Architecture）では，イベント通知の内容がイベント発生源となるアプライアンスの属性と状態を通知するもので，イベント通知先のアプライアンスが，その情報を解釈して，どうリアクションするか（アプリケーションのロジックとなる部分）を実装時に決定しなくてはならない．つまり，イベント通知先のアプライアンスは予めそのイベント通知を知っているという前提で，その対応を予め

[1]パケットサイズの比較には ethereal (http://www.ethereal.com/) を使用

コード内に実装されていないといけないということである．そのためには，予めイベントの発行者と受信者がインタフェースに同意を得ていなければならないという制約が生まれてしまう．もしくは，両者のインタフェースを結ぶためのプログラムを作成する必要が発生してしまう．イベントを通知された後の処理をどうするかは，ユーザによって異なると想定できるので，このような静的にロジックを設計に組み込んでしまうというアプローチは，ホームコンピューティング環境には向いていないと言える．

しかし，このイベント通知の部分を，今回の REST の実装では，イベント通知としてイベント通知先に取るべき状態を POST するという簡潔な手法ことによって実現している．これによって，接続するだけでその二者間の関係の意味を定義することが可能である（図 3.5 参照）．

また，統一のアドレス空間を使用することによって，アプライアンスという粒度ではなく，アプライアンスの持つリソース，さらにそれの取りうる状態に至るまで URI で一意に識別することが可能であることが，上記のようなイベント通知接続の実現に寄与していると考えられる．

今回の実装では言及しなかったが，今後，コンテキストなどの情報を扱えるようになった場合，REST を考慮すれば，それらも一意な URI で，デバイスの属性と同じように操作することが可能になると考えられる．これによって，揮発的なコンテキスト情報と，デバイスのベンダによって静的に用意されている永続的な情報とを，性質の違いを意識することなく検索できるようになる．

4.2.2 相互運用性

REST 導入によるアプライアンス実装における制約は，HTTP による入出力の機構を備えることと，そのコマンドの意味を正確に把握することのみである．このようなシンプルなアーキテクチャを採用することによって，アプライアンスの開発者間における事前了解事項（consensus）を最小限に抑えることができたと言える．このため，基本的な要素のみに対する制約しか存在しないため，各コンポーネント間の疎結合が実現でき，独立したアプライアンスの開発が可能である．また，SOAP のように多くの機能を仕様で定めていないため，2.1.3 節で示したバージョン互換の問題も起こりづらいと考えられる．

また，アプライアンス間のインタフェースは，HTTP というシンプルなメカニズムにより実現しているため，1.2.1 節で述べた人工的な依存の部分を極力減らすことに成功している．また，実依存の部分は，明確な意味を持つ HTTP コマンドによって表現されるため，標準として受け入れらやすいと考えられる．ホームコンピューティング環境におけるデファクトスタンダードを目指したときに，それを成し遂げやすいとも考えられる．このことは，WWW における HTTP の立場が証明している．

また，HTTP というプロトコルのみ依存しているため，実装言語に関しての制限がない点も，標準として各ベンダに導入する際の障壁を低くすることに貢献していると考えられる．

4.2.3 スケーラビリティ

SENCHA の想定している環境では，PAC となるパーソナルデバイスと，環境に存在するスマートアプライアンスの二種類があるが，PDA や小型のノートパソコンなど

比較的豊富なリソースを期待できるパーソナルデバイスに対して，スマートアプライアンスのリソースは，あくまで本来の機能の付随的な意味合いで組み込みシステムを搭載するという意味を考慮すると，豊富なリソースを期待できない．そのため，環境全体を考えると，一番のボトルネックとなるのは，スマートアプライアンスのリソースの制約であると考えられる．

この背景を考慮すると，REST を用いたアプローチの場合，複雑なデータの処理をアプライアンスに課すこともなく，RDF ファイルの解釈という最もオーバーヘッドとなる処理を，パーソナルデバイス上に集中させることにより，最もスケーラブルなアーキテクチャとなっていると考えれる．

しかし，REST におけるスケーラビリティの実現の根幹にある制約は，ステートレスな通信を行うという部分にある．本論文では，言及しなかったが，ホームコンピューティング環境では，デバイスを利用するユーザが必ず一人であるとは限らず，そのため，複数のユーザが存在する場合，操作が競合するのを防ぐために使用中のデバイスを占有状態にするというアプローチが考えられる．その場合，アプライアンス側（サーバ）で誰が使用しているかを管理しなくてはならず，この点では，REST のモデルに違反してしまっている．つまり，スケーラビリティを得られないということである．しかし，ホームコンピューティング環境におけるスケーラビリティは，WWW のようなグローバルなものを想定しておらず，また，占有状態にあるのなら，同時に使用するのは一人だけなので，スケーラビリティは重要でないというようにも考えられる．WWW というドキュメントベースの分散システムと，ホームコンピューティング環境のコントロールベースの分散システムとの性格の違いを考えると，ユーザの情報をサーバ側で管理しなくてはならない点など，REST が適切でない点も存在する．

4.2.4 ユーザデバイスを介さない接続

単純なインタフェースのみを提供し，かつ，そのインタフェースにはアプリケーションのロジックを含まず，接続するコネクションに意味を持たせることによって，ランタイムにおけるユーザの自由な接続が可能である．また，直接アプライアンス同士のコネクションを確立することが可能なため，ユーザがその環境に存在しなくてもアプライアンス間の関係を保持することが可能である．もし，関係が永続できるならば，例えば，違うユーザがその関係にアクセスし，どういう意味を持たしているかを検索できるメカニズムを実装できるかもしれない．そうすれば，他ユーザの利用履歴をすることによって，自分のコンフィギュレーションをより発展させることができるということが考えられる．

4.3 SENCHA の将来課題

4.1，4.2 節では，REST を SENCHA に導入することの利点を導き出すことができた．しかし，必ずしも全てが解決されたわけではなく，解決しなければならない問題も存在する．

4.3.1 ボキャブラリの定義

SENCHA の本来の目的のひとつである,「ユーザの設定した環境をどこでも再現する」ことを達成するためには，ボキャブラリの定義が必要であると考える．現状の実装では，イベント通知接続の情報保持に具体的な IP アドレス，リソース名を使用しているが，これをさらに抽象的なアプライアンスのタイプ，リソースの意味などのように汎用的な情報を用いて表現することができれば，アプリケーションテンプレートとして，他の環境でも同じ環境を再現できるようになる．スタンダードなボキャブラリを使用することも可能だが，それだけでは足りない可能性もあることも考慮しなくてはならないと考えられる．

4.3.2 リソースの粒度

また，本論文でサービスリソースというものを定義したが，必ずしもこの粒度のリソースを定義する必要はないと言える．3.2.1 節のサービスリソースの説明で挙げた例の場合，２つのロジックを含んでいる．このロジックをサービスのサブセットとして表現するのではなく，単体のリソースとして定義することによって，より汎用的なアプリケーションの構築が可能になると考えられる．例えば，インターバルロジックを monitorable なリソースとして定義し，セットした秒数ごとにイベントを発行するものと解釈すれば，他の機能リソースとの連携が可能になる．例えば，ライトと連携させることによって，セットした秒数ごとにライトを明滅させるといったアプリケーションも可能である．また，3.2.1 節のサービスの例も，インターバルロジック，チャンネルセットロジック，チャンネル機能リソースの３つの独立したリソースを組み合わせることで同じサービスを表現することができる．このように粒度を最小のものに設定すれば，ユーザによるアプリケーションの開発の自由度をさらに増すことができる．しかし，アプリケーション開発の自由度と，それをするための手間はトレードオフの関係にあるため，どのような粒度を設定すべきかは一概には言うことができない．

4.3.3 データ転送

今回のデータ転送における実装では，データシンクにソースの URI を注ぎ込む，もしくは，データソースにシンクの URI を設定をすることによってコネクションを設定した．この仕様の前提となっているのは，データの再生コントロールが一方に集中しているということである．ソースが単純にデータを WEB サーバ上で公開している，もしくは，ストリーミング放送のような常に再生されているデータをシンクが受信し，シンクとなるデバイスの機能を使ってコントロールする場合に使用が可能である．しかし，ソース側からの転送と，シンク側での受信・再生の両方にコントロールが存在する場合のように，二者間での同期が要求される場合もある．

スキームと MIME タイプによるメディアデータの分類と，HTTP によるシンプルな操作によって，データ転送接続を容易に行うことはできるが，その接続に対して二者間の同期を取るという要求が出てくる場合，刻一刻と変化するストリーミングのシーケンシャルなイベントを管理しなくてはならない．そのため，REST の単純なメカニズムでは対応仕切れない．REST の対応できるデータは，断続的なものに限られるた

め，連続的なデータを扱うには Speakeasy のモバイルコードによる方法 [9] のように別の機構を導入することが良いと考えられる．

第5章　関連研究

5.1　Jini

Jini[26] は Sun Microsystems によって提唱されている仕様で，UPnP とコンセプトを同じにするものであるが，実装言語は Java に限定されている．また，デバイス間のコミュニケーションは JavaRMI のメカニズムを利用しており，デバイスによって提供されるサービス（JavaRMI のスタブ）は Lookup サーバというディレクトリサーバによって管理されている．サービスを利用したいクライアントは，Lookup サーバにそのサービスを問い合わせれば良いが，登録されているサービスを利用するためには，そのインタフェースを予め知っている必要があるため，サービス提供者と，サービス利用者が強く依存してしまっているという問題がある．

5.2　ICrafter

ICrafter[4] は，スタンフォード大学における Interactive workspace プロジェクトにて研究されているサブプロジェクトのひとつで，環境に偏在するデバイスのアグリゲーションのためのフレームワークである．特に UI の自動生成に着目しているシステムである．

UI の自動生成に関しては，SENCHA と同じくデバイスから提供されるサービス記述を解釈することによって行っている．これによって，個々のコントロールのための UI，複数のデバイスを同時にコントロールするための UI を自動生成することができる．しかし，複数のデバイスのコントロールは，パターンが予めインタフェースによって決められている．例えば電源に関するインタフェースには，PowerSwitchInterface という Java のインタフェースに相当するものが用意されており，これを実装しなくてはならない．UI generator は，この制約を前提に,「すべての機器の電源をオフにする」という UI を生成することができる．他の例としては，DataConsumer インタフェースを実装しているデバイスと，DataProducer インタフェースを実装しているデバイスを接続するなど，デバイス間の接続は設計段階でインタフェースによって決められてしまっている．そのため，予めインタフェースとして用意されていない組み合わせには対応できないという問題がある．例えば，テレビの電源を付けたら，天井のライトを消す，もしくは，テレビの電源を付けたら，暗くするという組み合わせは不可能である．他にも，ユーザの要求によって組み合わせは様々なものが考えれることを考慮すると，予めインタフェースによって組み合わせのパターンを決めてしまうアプローチは 1.2.1 節で述べたエンドユーザプログラミングの要件を満たしていない．

5.3 SpeakEasy

SpeakEasy[9] は，Xerox の Palo Alto 研究所で行われているプロジェクトで，Jini のようにモバイルコードを用いた手法によって，ネットワーク内にあるコンポーネント間の高い相互運用性を実現しようとしたフレームワークである．Jini における問題点は，5.1 節にて説明したとおり，クライアントが予め利用するサービスのインタフェースを理解していなくてはいけない点である．SpeakEasy では，コンポーネント間のデータ転送においてこの問題を解決し，最小限の syntax を事前了解事項とし，semantic な情報を自由に定義できるフレームワークを実現している．しかし，コンポーネント間の通信はメディアデータの転送に限られ，イベント通知などの通信はサポートしていないため，コンポーネント間の関連付けなどは自由に行えない．

第6章　結論

本論文では，SENCHA というデバイスアグリゲーションを目的としたミドルウェアに REST のアーキテクチャ的スタイルを導入することによって，ホームコンピューティング環境における REST の有用性を検証した．SOAP による実装との比較により，REST を用いる方法は軽量かつ高速であることがわかり，かつ，デバイス間のインタフェースを固定された意味を持つ HTTP による少数セットのコマンドによって実現することによって，ユーザによる自由な接続を動的に行うことを可能とした．このことより，ホームコンピューティング環境という分散システムの環境においても，REST という簡潔なアーキテクチャ的スタイルは有用であると考えられる．Newmarch[10] も，本研究と同様に UPnP に REST を導入することを提案しており，SOAP と REST の定量的な比較により REST の有用性を示している．

しかし，必ずしも REST がすべての状況に適しているとは言い難いという点も考慮しなくてはならない．例えばコンテキストな情報を提供するアプライアンスを考慮する場合，発行されるイベントが必ずしも断続的なものであるとは限らない．今回の実装においては，連続的なイベント発行に関しては言及しなかったが，シーケンシャルなイベントを解釈する機構をアプライアンスに想定することは REST の観点からは不向きである．言葉を変えればイベントは，すべて各々の発行元で，断続的な情報として提供されるべきである．例えば，押下型のボタンを考えた場合,「押し続ける」という状態があるかもしれない．このような場合は,「押されている」という状態のイベントを連続的に送るのではなく,「pushed」「released」の 2 つの状態で表現することが REST には必要になってくる．このボタンと連携させたいアプライアンスには，ボタンを「押した」というイベントが必要な場合と,「押されている」というイベントが必要な場合が考えられる．REST-enabled なアプライアンスがこれらの要件に対応するためには，イベントの種類を複数視点から用意することも必要である．これらは REST-enabled SENCHA システムの将来課題として提案したい．そのためには，イベントの種類の分析，ボキャブラリの定義などの作業が必要とされると考えられる．

また，ホームコンピューティング環境を実現するためのミドルウェアやフレームワークは，本研究以外でも多数なされている．その中でも UPnP や HAVi，Jini のような標準的なフレームワークを連携するというアプローチも存在するが，このアプローチでは，結局，各々のフレームワークが提供する機能を全て連携することができず，導入のコストもかかるという欠点がある．複数のプロトコルや仕様を導入するよりも，単一の標準を導入する方が効率が良いと言える．単一の標準となる仕様を策定することは，必ずしも容易なことではなく，REST に証明されているようにシンプルなものがデファクトスタンダードに成り得ることが WWW によって証明されている．WWW に似たシステム要件を持つホームコンピューティング環境において，REST という簡潔なアーキテクチャを導入するとういアプローチは一理あると言える．

謝辞

本研究のために御指導頂いた中島達夫教授を始め，グループリーダーである石川広男氏に深く心より感謝いたします．また，同じ研究室の下で，共に切磋琢磨することができた仲間達に深く心より感謝いたします．

参考文献

[1] Mark Wiser: "The Computer for the 21st Century", *Scientific American*, 265(3):94–104, Sep. 1991.

[2] Hiroo Ishikawa, Yuuki Ogata, Kazuto Adachi, and Tatsuo Nakajima: "Building Smart Appliance Integration Middleware on the OSGi Framework", *In proceedings of The 7th IEEE International Symposium on Object-oriented Realtime distributed Computing* , May. 2004.

[3] 安達一斗，生形裕貴，石川広男: "SENCHA: ユーザの近傍の情報アプライアンスの協調を支援するミドルウェア "，第 7 回プログラミングおよび応用のシステムに関するワークショップ (SPA2004), Jun. 2004.

[4] Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd: "ICrafter : A Service Framework for Ubiquitous Computing Environments", *In Proceedings of UBICOMP 2001*. Atlanta, Georgia, USA, Sep. 2001.

[5] Irene Mavrommati, Achilles Kameas: "The evolution of objects into hyper-objects: will it be mostly harmless?", *Personal and Ubiquitous Computing*, ACM. vol.7, no. 3-4. ISSN 1617-6909, pp.176-181, July. 2003.

[6] Roy T. Fielding: "Architectural Styles and the Design of Network-based Software Architecture", PhD thesis, Univ. of California, Irvine, 2000.

[7] Roy T. Fielding and Richard N. Taylor: "Principled Design of the Modern Web Architecture", *In proceedings of 22nd International Conference on Software Engineering (ICSE'2000)*, Limerick, Ireland, June. 2000.

[8] Roy T. Fielding and Richard N. Taylor: "Principled Design of the Modern Web Architecture", *ACM Transactions on Internet Technology*, Vol2, No. 2, pp.115-150, May 2002.

[9] W. Keith Edwards, Mark W. Newman, Jana Z. Sedivy: "The case for Recombinant Computing", Xerox Palo Alto Research Center Technical Report CSL-01-1, April 20, 2001.

[10] Jan Newmarch: "A RESTful Approach: Clean UPnP without SOAP", *IEEE Consumer Communications and Networking Conference*, Jan 2005.

[11] OSGi Alliance
`http://www.osgi.org/`

[12] UPnP Forum
http://www.upnp.org/resources/whitepapers.asp

[13] RDF: Resource Description Framework
http://www.w3.org/RDF/

[14] Web Services Description Language (WSDL) 1.1
http://www.w3.org/TR/wsdl

[15] Jena 2 - A Semantic Web Framework
http://www.hpl.hp.com/semweb/jena2.htm

[16] Apache Axis
http://ws.apache.org/axis/

[17] Apache Tomcat
http://jakarta.apache.org/tomcat/

[18] SOAP Specifications
http://www.w3.org/TR/soap/

[19] X10
http://www.x10.com/

[20] Notation3
http://www.w3.org/DesignIssues/Notation3

[21] N-Triples: W3C RDF Core WG Internal Working Draft
http://www.w3.org/2001/sw/RDFCore/ntriples/

[22] CORBA IIOP Specification http://www.omg.org/technology/documents/formal/corba_iiop.htm

[23] DCOM http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomtec.asp

[24] URI
http://www.w3.org/Addressing/

[25] Genaral Event Notification Architecture (GENA)
http://www.upnp.org/download/draft-cohen-gena-client-01.txt

[26] Jini
http://www.sun.com/software/jini/specs/

[27] HAVi (Home Audio Video Interoperability)
http://www.havi.org/

[28] SSDP specification
http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt

[29] Xerces Java Parser
http://xml.apache.org/xerces-j/

付 録A　WSDLの例

A.1　ライトアプライアンス

```
<?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions targetNamespace="http://localhost:8080/axis/services/Light"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:impl="http://localhost:8080/axis/services/Light"
    xmlns:intf="http://localhost:8080/axis/services/Light"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
- <wsdl:message name="getPowerResponse">
    <wsdl:part name="getPowerReturn" type="xsd:string" />
  </wsdl:message>
- <wsdl:message name="setPowerRequest">
    <wsdl:part name="value" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="getPowerRequest" />
- <wsdl:message name="setPowerResponse">
    <wsdl:part name="setPowerReturn" type="xsd:string" />
  </wsdl:message>
- <wsdl:portType name="Light">
- <wsdl:operation name="setPower" parameterOrder="value">
    <wsdl:input message="impl:setPowerRequest" name="setPowerRequest" />
    <wsdl:output message="impl:setPowerResponse" name="setPowerResponse" />
  </wsdl:operation>
- <wsdl:operation name="getPower">
    <wsdl:input message="impl:getPowerRequest" name="getPowerRequest" />
    <wsdl:output message="impl:getPowerResponse" name="getPowerResponse" />
  </wsdl:operation>
  </wsdl:portType>
- <wsdl:binding name="LightSoapBinding" type="impl:Light">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="setPower">
    <wsdlsoap:operation soapAction="" />
- <wsdl:input name="setPowerRequest">
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       namespace="http://soap.test.sencha" use="encoded" />
  </wsdl:input>
- <wsdl:output name="setPowerResponse">
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       namespace="http://localhost:8080/axis/services/Light" use="encoded" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="getPower">
    <wsdlsoap:operation soapAction="" />
- <wsdl:input name="getPowerRequest">
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://soap.test.sencha" use="encoded" />
  </wsdl:input>
```

```
- <wsdl:output name="getPowerResponse">
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://localhost:8080/axis/services/Light" use="encoded" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:service name="LightService">
- <wsdl:port binding="impl:LightSoapBinding" name="Light">
    <wsdlsoap:address location="http://192.168.123.63:8080/axis/services/Light" />
  </wsdl:port>
  </wsdl:service>
  </wsdl:definitions>
```