NAIST-IS-MT0651023

修士論文

スケーラビリティを考慮した センサ駆動アプリケーションフレームワーク

大西　洋司

2008年2月7日

奈良先端科学技術大学院大学
情報科学研究科 情報システム学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
修士 (工学) 授与の要件として提出した修士論文である。

大西　洋司

審査委員：

松本 健一 教授　　（主指導教員）
伊藤　 実 教授　　（副指導教員）
門田 暁人 准教授　（副指導教員）
中村 匡秀 准教授　（神戸大学）

スケーラビリティを考慮した センサ駆動アプリケーションフレームワーク*

大西　洋司

内容梗概

近年のセンサ機器の発達により，センサの状態を監視し，センサの値があらかじめ指定した条件に一致したときに何らかの処理を行うというセンサ駆動サービスが普及しつつある．しかし，利用するセンサの個数が増加すると，多数のセンサを同時に扱うアプリケーションが複雑になるというセンサとアプリケーションの密結合の問題や，センサから取得されるデータが膨大になりネットワークやアプリケーションの負荷が増大するといったスケーラビリティの問題点が発生する．

本論文では, これらの問題を解決するスケーラビリティを考慮したセンサアプリケーションフレームワークを提案する．提案フレームワークでは，センサにサービスレイヤを持たせ，センサ固有の情報を扱い，処理をすべて行わせることで，アプリケーションの密結合の問題を解決した．それに加え，Publish/Subscribe 型のメッセージ交換パターンをベースとした通信を用いることで通信やアプリケーションの負荷を軽減させることができた．また，提案フレームワークを実装し，上記の問題点が解決できていることを実験により確認した．

キーワード

センサ駆動サービス, センサミドルウェア, イベント駆動, 疎結合, 負荷分散

*奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 修士論文, NAIST-IS-MT0651023, 2008 年 2 月 7 日.

An Implementation of Scalable Sensor Application Framework*

Yoji Onishi

Abstract

Many integrated services with sensors and its appliances become common in our daily life. In most of those services, the application and sensors are tightly-coupled. This causes the implementation of the application becomes more complicated. Moreover, network load between the application and the sensor and processing load to evaluate the sensor data increase severely. As the number of sensors to connect increases, the problems become more serious.

This paper proposes a scalable sensor application framework. Using a standardized API, applications can access any sensors without implementing any sensor-specific procedure. Furthermore, the application delegates a part of evaluation process of the trigger conditions to the sensor. Due to the delegation, network load is minimized because the application communicates with the sensor only when the status of the registered condition is changed. Using these methods, we evaluate this framework by measuring network load.

Keywords:

sensor driven service, sensor middleware, event driven, loose coupling, load balancing

*Master's Thesis, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT0651023, February 7, 2008.

関連発表論文

国際会議

1. Yoji Onishi, Hiroshi Igaki, Masahide Nakamura, and Ken-ichi Matsumoto. A Scalable Sensor Application Framework Based on Hierarchical Load-Balancing Architecture. In *Proceedings of the IASTED International Conference on Software Engineering (IASTED SE 2008)*, pp.37-42, February 2008.

研究会・シンポジウム

1. 大西洋司, 前島弘敬, 西澤茂隆, 田中秀一郎, 中村匡秀, 松本健一. 時間駆動型 Web サービス呼び出しフレームワーク WS-Schedule Manager の提案と実装. 電子情報通信学会技術研究報告, Vol.106, No.578, pp.459-464, March 2007.

その他の発表論文

国際会議

1. Yasutaka Kamei, Shinsuke Matsumoto, Hirotaka Maeshima, Yoji Onishi, Masao Ohira, and Ken-ichi Matsumoto. Analysis of Coordination between Developers and Users in the Apache Community. In *Proceedings of the International Conference on Open Source Systems (OSS2008)*, September 2008, (to appear).

研究会・シンポジウム

1. 前島弘敬, 大西洋司, 中村匡秀, 松本健一. BPEL ワークフローに着目した連携 Web サービスの応答速度・稼働率の見積もり手法. 電子情報通信学会技術研究報告, Vol.106, No.578, pp.465-470, March 2007.

2. 大蔵君治, 大西洋司, 川口真司, 大平雅雄, 飯田　元, 松本健一. メールスレッドのクラスター分析による OSS プロジェクトのアクティビティ予測手法. 電子情報通信学会技術研究報告, vol.107, No.275, SS2007-37, pp.41-46, October 2007.

3. 前島弘敬, 柗本真佑, 亀井靖高, 柿元　健, 大西洋司, 大平雅雄, 松本健一. コーディネータのコミュニティ媒介性の評価指標の提案. 情報処理学会シンポジウム, Vol.2007, No.11, pp.71-76, November 2007.

目 次

図 目 次

表 目 次

1. はじめに

組み込み技術，ネットワーク技術の進歩に伴い，多様な家電製品や電気機器が誕生し，人々の生活に深く浸透している．それらの機器はより高性能，高機能に進化しているだけでなく，複数の機器同士を連携させて動作する機能が普及してきた [1, 2, 3]．また，機器同士の連携に限らず，センサと機器を組み合わせてより知的に動作する連携機能の開発・普及が進んでいる．ユビキタスコンピューティングの分野では特に，ユーザの挙動や環境の変化をセンサにより検知し，事前に登録された条件に適合した場合にサービスを駆動するセンサ駆動型のアプリケーションの開発が実際に行われている．

例えば，ホームネットワークシステムやビル管理システムでは，センサと組み合わせて動作することを前提とした機器が数多く運用されている．人感センサを利用し，人が近くに来るとライトをつけるセンサライト [4]，人が近づくとドアを開ける自動ドア [5] や，温度センサを利用して空調を適度に保つエアコンの自動運転制御 [6]，手をかざすことで水道の蛇口が開く自動水栓 (automatic sensor faucet)[7] などが存在する．また，より高度な例では，ユーザの血圧や薬を定期的に摂取しているか，倒れたりしていないか等のデータをセンサによって取得し，家族など介護を行っているユーザや救急車を呼ぶといった，ユーザの状態に適した行動を選択・決定し実行する生活支援システムなどが提案されている [8]．このように，ユーザの周囲に複数のセンサとそのセンサを利用する複数のユビキタスアプリケーションが配備されるようになりつつある．

このようなセンサ駆動型アプリケーションにおいてよりきめ細やかなサービスをユーザに提供しようとした場合，機器に接続するセンサを増やし，周囲環境の情報を多数集めることで改良することができる．しかし，現状では機器に接続するセンサを増やした場合，2 つの問題点が発生するため，実現が難しい．

1 つ目の問題は，センサと機器が密に結合している点である．センサを扱うアプリケーションがセンサの値を得ようとした場合，センサへのアクセス方法やセンサの値の解釈といったセンサ固有の処理をアプリケーション内に作り込む必要がある．そのため，センサが増加するとアプリケーションの複雑さが増大し，センサの追加・変更時のアプリケーションの改修コストが大きくなってしまう．

2 つ目の問題は，接続するセンサの数を増加させることに対するスケーラビリティが低い点である．センサが検知する値は，環境の変化に伴って変化し続ける．センサを利用するアプリケーションは絶えずその値を監視し続けている必要があるため，センサ数の増加に伴い，センサとアプリケーション間の通信量やセンサデータを処理するアプリケーションの負荷も増大する．

そこで，本論文では多数のセンサを利用するようなユビキタスサービス実現において考慮が必要なこの密結合の問題 (問題 1) とスケーラビリティの問題 (問題 2) に対応したスケーラブルなセンサアプリケーションを構築可能にするフレームワークを提案する．

まず問題 1 に対応するために，センサデバイスをサービスレイヤと呼ぶ層で包み込む．このサービスレイヤはセンサ固有のアクセス方法や値解釈に関するロジックを Web サービス技術 [9] を利用した標準的な API に変換するラッパーで，センサとアプリケーション間の疎結合を実現する．ユビキタスサービスは，このサービスレイヤを通じて，各センサが検知可能なプロパティの情報と，そのプロパティの値を取得し，アプリケーション内でセンサ固有のロジックを記述することなく多様なプラットフォームから簡単に利用することが可能となる．

問題 2 に対しては，サービスを駆動するための条件がセンサの示す値の閾値から構成されるケースが多いことに着目し (例えば温度計の場合，温度が 27 度以上になったら冷房の電源を入れるなど)，この種の条件判定をアプリケーションではなく各センサのサービスレイヤに委譲する仕組みを作成した．アプリケーションはサービスレイヤを通して，センサごとに条件を登録 (*subscribe*) する．各サービスレイヤは登録された条件にもとづいてセンサの値を監視し，条件が成立したとき，または不成立になったときのみ，それぞれ真/偽という返答をアプリケーションに通知 (*publish*) する．このような Publish/Subscribe 型のメッセージ交換パターン [10] を応用したアーキテクチャにより，センサとアプリケーション間の通信が登録された条件が成立/不成立のときしか行われなくなる．結果として，問題 2 で述べた通信コストの問題は大いに改善される．また，条件判定の部分がセンサに委譲されるため，アプリケーションの負荷も低減される．

さらに，提案するサービスレイヤを利用することで，複数のセンサの取る閾値

を組み合わせた複雑な条件を検知するメタセンサを容易に実装することが可能となる．例えばエアコンの自動運転制御では,「室温が 28 度以上かつ湿度が 50%以上のときに強運転を行う」といった，複数のセンサをまたがった条件を記述することができる．

本論文では，提案するフレームワークを用いてセンサ機器と家電機器とを対象としたセンサ駆動サービスを提供するアプリケーションを開発し，センサの入れ替えやユーザ要求の変化，サービス内容の修正への対応が可能であることと，センサとアプリケーションとの間の通信量がどの程度削減されるかをケーススタディとして行った．

以降，第 2 章でセンサ駆動サービスの定義と問題点を説明し，第 3 章で提案フレームワークを説明する．第 4 章で実際に実装されたフレームワークの負荷を従来法と実験的に比較し，第 5 章で考察を行い，第 6 章でまとめを行う．

2. 準備

2.1 センサ駆動サービス

センサ駆動サービスとは，環境の状態を数値化する**センサデバイス** *(sensor device)* から値を取得し，その値があらかじめ決めておいた**条件**に一致したときに，あらかじめ**決めておいた機器を駆動**させるサービスである．センサデバイスの種類は多様にあり，温度センサ，人感センサ，赤外線距離センサなどがある．センサデバイスごとに取得できる情報は異なり，温度センサであれば温度の値，人感センサであれば人の存在の有無，赤外線距離センサであればセンサからセンサに近い物体への距離が得られる．それぞれのセンサは温度や距離といった単一の**プロパティ**を保持し[1]，そのプロパティが取得できる値の範囲内で，そのプロパティがとる**環境属性値** *(environmental value)* を監視する．

このようなセンサ駆動サービスについての定義と詳細な説明を，次節以降で述べる．

2.2 センサの定義

1 つのセンサ駆動サービスで用いられるセンサデバイスを式 (1) と定義する．

$$s_i \in S(1 \leq i \leq n) \tag{1}$$

式 (1) において，s は単一のセンサデバイスを表し，S は 1 つのセンサ駆動サービスで用いられているセンサデバイスの集合を表す．各センサデバイス s_i はそれぞれ 1 つのプロパティ p を持ち，プロパティ p に基づいた値を取得することができる．またプロパティ p はそれぞれ型制約 t_v と呼ばれる，センサが取りうる値の種類（例えば，整数型をとる，真偽のいずれかをとるなど）とその範囲の制約を持ち，型制約 t_v に基づいたセンサ値 v を取る．

[1]センサによっては，1 つのセンサに複数のプロパティを保持するものも存在する．そのような場合，そのセンサは 1 つのプロパティを持つ複数のセンサの集まりとみなすことができる．

表 1 センサにおける型制約 t の種類

型名	型	制約の要素	表記の例
$integer$	整数型	最小値と最大値をとる	$int\{0, 100\}$
$float$	浮動小数点型	最小値と最大値をとる	$float\{3.5e-2, 7.0e-3\}$
$string$	文字列型	文字列をとる	str
$enumerate$	列挙型	選択できる要素の組のうち 1 つをとる	$enum\{$“$weak$”, “$middle$”, “$high$” $\}$
$boolean$	ブール型	$true$ もしくは $false$ をとる	$bool$

一般的に，環境属性値はセンサ値 v とは異なった数値をとる．異なるベンダから開発されたセンサはそれぞれ仕様が異なるため，センサ値 v を人間が解釈しやすく，条件記述において用いやすい表現である環境属性値 e に変換するための機能をセンサに合わせて開発しなければならない．すなわち，多くのセンサ s は式 (2) に示すような，測定したセンサ値 v を環境属性値 e に変換するための関数 f を持つ．

$$e = f(v) \tag{2}$$

また，センサ値 v に型制約があるのと同様に，センサから得られたセンサ値 v をセンサ固有の関数 f で変換した環境属性値 e にも型制約 t_e がある．これらの型制約の種類を表 1 に示す．

Phidgets[11] の温度センサの例を考える．Phidgets の温度センサ s は温度というプロパティ p を持つ．プロパティ p は型制約 $t_v = int\{40, 700\}$(整数型をとり，40 から 700 の間の値を取ることを表す) を持ち，その制約の中でのセンサ値 v をとる．もしアプリケーションが温度センサを用いて温度 p を取得しようとした場合，まずアプリケーションは温度センサからセンサ値 v を取得し，式 (3) のような値変換の関数によってセンサ値から環境属性値に値を変換する必要がある．

$$e = f(v) = \frac{v - 200}{4} \tag{3}$$

また，式 (3) で得られた環境属性値 e は，型制約 $t_e = int\{-40, 125\}$ を満たす．

環境属性値 e は，センサのプロパティ p における単位系による制約と，センサデバイスの性能による制約といった 2 つの制約を受けることで，とりうる値の範囲がセンサごとに変化する．例えば，センサのプロパティ p が温度の場合，単位系による制約として $-273°C$ 未満にはならないといったものがある．Phidgets[11] の温度センサであれば，センサ値の型制約 t_v と変換関数 f から $t_e = int\{-40, 125\}$ という制約があることが分かる．

2.3 センサ駆動条件の定義

センサアプリケーションにおけるセンサの主な役割は，現在の環境属性値を取得することである．アプリケーションはあらかじめセンサ駆動サービスに登録された駆動条件に基づいて機器を駆動させるために，登録されたセンサを監視し，環境属性値の変化を監視している．駆動条件が真か偽になったかどうかを評価するために，アプリケーションは継続的にセンサと通信を行う．

本論文では，環境条件を式 (4) で定義する．

$$E = \{e_1, e_2, ..., e_n\} \tag{4}$$

E は環境属性の組を表し，単一の環境属性 e の組で表される．また，$atom(e)$ を単一の環境属性値 e に依存した論理式と定義し，環境属性の組 E の環境条件を $condE$ と定義する．また，すべての環境条件 $condE$ は $atom(e)$ の組み合わせで定義される，図 1 のバッカス・ナウア記法 (BNF)[12] で表される．これらの e，E，$atom(e)$，$condE$ の関係を表 2 に示す．

センサ駆動サービスは環境条件と動作の組で表される．例えば，センサライトサービスは式 (5) のような条件と動作を持っていると考えられる．

$$\begin{aligned} condE: &\ \text{"}human\_detect == true \ \&\& \ brightness < 10\text{"} \\ action: &\ \text{"}turn\_on\_the\_light\text{"} \end{aligned} \tag{5}$$

$condE$ ::	$condE$ && $condE$	(AND)
	$condE$ || $condE$	(OR)
	! $condE$	(NOT)
	$(condE)$	
	$atom(e_i)$	

図 1 環境条件の BNF による表現

表 2 環境属性値の実環境における値と計測数値との違い

	単位環境属性	環境属性の組
実環境の環境属性値	e	E
計測・記述される値	$atom(e)$	$condE$

式 (5) では，$human\_detect$ センサが $true$ の値をとり (人を検知し)，brightness センサが 10 ルクス未満の値をとる (暗くなる) と，動作 "$turn\_on\_the\_light$" を呼び出すという条件を意味している．

一般的に，上記のような条件文における動作 (action) は家電統合サービスや他の API 呼び出しなどのメソッド呼び出しの組を持つアプリケーションを呼び出すことによって実行する．本論文では，動作条件の登録部分のみに焦点を当て，呼び出した後の動作についての詳細には焦点を当てないこととする．

2.4 条件の記述

センサ駆動サービスでは，登録された機器を駆動させるために，機器の駆動条件があらかじめ登録されており，センサから得られた値をその条件文に代入することによって条件文の真偽を判定している．

まず，1 つのセンサに対する条件文を考える．センサに対する条件文を記述する際は，そのセンサから得られたセンサ値が，あらかじめ決められた値と比較することで条件文の真偽が判定される．単一のセンサに対する条件文の記法を式 (6)

表 3 利用できる 2 項演算子とその意味

2 項演算子	意味
$<$	左辺が右辺よりも小さい
$>$	左辺が右辺よりも大きい
$<=$	左辺が右辺よりも同じか小さい
$>=$	左辺が右辺よりも同じか大きい
$==$	左辺と右辺が等しい
$!=$	左辺と右辺が異なる
eq	左辺と右辺も文字列が一致する

表 4 利用できる関係演算子とその意味

関係演算子	意味
x && y	x と y の両方が成り立てば成り立つ
$x \mid\mid y$	x と y のうち一方が成り立てば成り立つ

に示す.

$$atom(e) = sensor\_name \;\; operator \;\; value \tag{6}$$

$sensor\_name$ は，今注目しているセンサの固有 ID を示し，$value$ は比較する値を示している．演算子 (operator) には表 3 に示す 2 項演算子が考えられる．センサでは，実際にこのような条件式にセンサ値を代入することで条件文を判定し，機器を駆動するかどうかを決定している.

複数のセンサについての条件文を記述するには，式 (6) で示すセンサごとの条件文を表 4 に示す論理演算子で組み合わせることで式 (7) のような記法で記述することができる.

$$condE = atom(e) \;\; operator \;\; atom(e) \tag{7}$$

なお，単一のセンサに対する条件文 $atom(e)$，複数のセンサに対する条件文

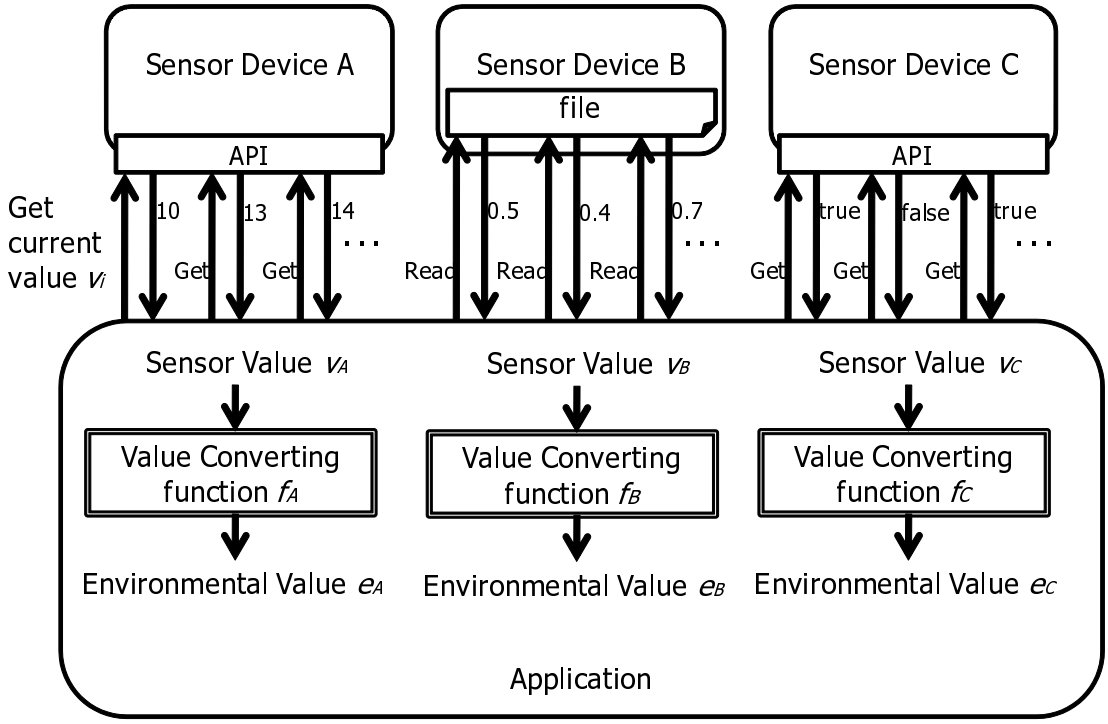

図 2 従来のセンサアプリケーション

$condE$ の詳細な記法については，付録 A で示した．

2.5 従来のセンサアプリケーションにおける問題点

従来のセンサ駆動サービスを用いたセンサアプリケーションの模式図を図 2 に示す．

図 2 において，センサデバイスからセンサ値を得る方法はセンサごとに異なる．センサデバイスに固有の API が用意されていたり，最新のセンサ値がファイルとして保存されており，そのファイルにアクセスすることでセンサ値を得られるといったセンサへのアクセス方法がある．また，センサ値の型制約 t_v がセンサデバイスごとに異なる点，センサ値 v を環境属性値 e に変換するための変換関数 f がセンサデバイスごとに異なる点のために，センサデバイスにアクセスするためのアプリケーションの仕様もセンサデバイスごとに異なる．これらのセンサデバイ

ス間の違いはアプリケーションの違いに直接的に影響を与えてしまう．そのため，アプリケーションの開発者は以下の 2 つの問題点に直面する．

問題 1: センサとアプリケーションの結合が密

問題 2: スケーラビリティが低い

アプリケーションで使われるセンサの種類は，そのセンサ駆動サービスの内容に依存する．サービスの駆動条件や振る舞いを更新したり，センサを交換したりしようとした場合，開発者はアプリケーションに含まれているセンサ固有の記述と条件を更新しなければならない．そのため，問題 1 に示すように，センサ固有の情報がアプリケーションに含まれていることから，アプリケーションの複雑さが増加し，センサ駆動サービスの柔軟なカスタマイズ性を妨げてしまう．

問題 2 の低いスケーラビリティは，アプリケーションのネットワークの負荷と条件判定処理の負荷に起因する．センサ駆動サービスの数と種類が増加するにつれ，1 つのセンサが多くのアプリケーションから使われることになる．図 2 に示すように，アプリケーションは継続的にセンサから値を取得しなければならない．アプリケーションはセンサで検知した値を監視し続け，センサ駆動サービスの駆動条件に一致する動作を実行するために環境条件を評価し続けなければならない．その結果，非常に多くのセンサが頻繁にアクセスされ，アプリケーションが多くの条件判定処理を行うため，ネットワークと条件判定処理のパフォーマンスが悪くなってしまう．

3 章以降では，これらの問題を解決するセンサアプリケーションフレームワークについて説明する．

3. 提案フレームワーク

3.1 キーアイデア

2章で述べた問題点を解決するために，提案フレームワークでは次の3つのキーアイデアを用いる．

1. センサへの標準的なインタフェースの提供
2. 条件判定ロジックをセンサへ委譲
3. サービスレイヤの階層化によるメタセンサの実現

提案フレームワークでは，センサ固有の情報であるセンサ値 v_i や変換関数 f_i をアプリケーションで扱うのではなく，新たにセンサにセンサ固有の情報を扱う**サービスレイヤ**を設置し，そこでそれらの情報をすべて扱うこととする．これにより，アプリケーションはセンサ固有の情報やロジックを知らずとも，環境情報 e_i のみを扱えるため，センサとアプリケーションの結合度を弱くすることができる．また，センサ固有の情報や処理だけでなくセンサ駆動サービスの条件判定もサービスレイヤに委譲する．このことにより，アプリケーションにおける条件判定の負荷やネットワークの負荷を低減することができるようになる．さらに，複数のセンサを用いる条件文を簡易に扱えるようにするため，メタセンサを提案する．

これらのキーアイデアについて，次節以降で説明する．

3.2 センサへの標準的なインタフェースの提供

サービスレイヤを用いた提案センサの構成を図3に示す．サービスレイヤでは，センサ s_i が持つセンサごとに固有の情報であるプロパティ名 p_i や型制約 t_{vi}，センサ値 v_i を保持し，これらの情報はサービスレイヤ内でのみ扱う．センサは，アプリケーションに対して，環境情報 e_i のみを共通インタフェース $getStatus()$ を用いて提供する．アプリケーションは，条件判断に不要な型制約 t_{vi} やセンサ値 v_i，変換関数 f_i を知ることなく環境情報 e_i のみを用いることができるため，セン

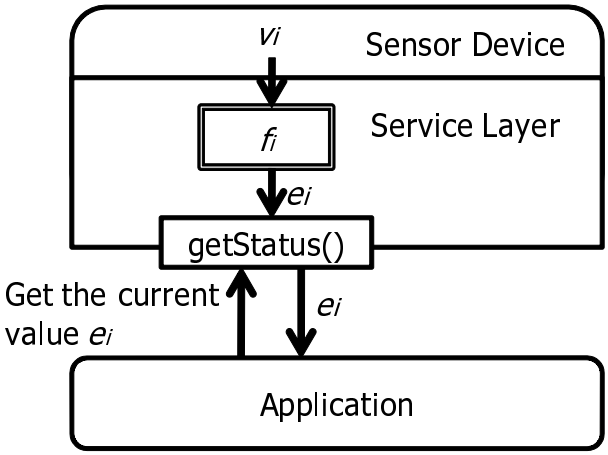

図 3 標準的なインタフェースの提供

サとアプリケーション間の依存性を弱くすることができる．また，インタフェースが共通であるため，センサを変更してもアプリケーションを再構築することなく同じアクセス方法を利用して値を取得することが出来る．

3.3 センサへの条件登録による条件判定処理の委譲

3.2 節における $getStatus()$ メソッドによって，センサ固有の情報は知らずともセンサの環境属性値を取得できるようになった．しかしながら，最新の環境属性値によって条件の真偽を判定する場合，絶えず $getStatus()$ メソッドを呼び出し，最新の環境情報値を取得し，条件文が成立するかをアプリケーションが監視しなければならない．この現在のセンサ値を取得するための継続的なアクセスによって，ネットワークやセンサの負荷が増大してしまう．提案フレームワークでは，センサから得られた環境属性値 e_i を用いた条件文の真偽値判定をサービスレイヤに委譲することでこの問題を解決する．

図 4 では，条件判断を行う処理を追加したサービスレイヤの処理の流れを示している．図 4 において，サービスレイヤに新たに条件を追加するためのメソッド $subscribe()$ を追加している．$subscribe()$ メソッドは，センサを利用するアプリケーションから，各センサがとりうる環境属性値を用いた条件文とその条件文の

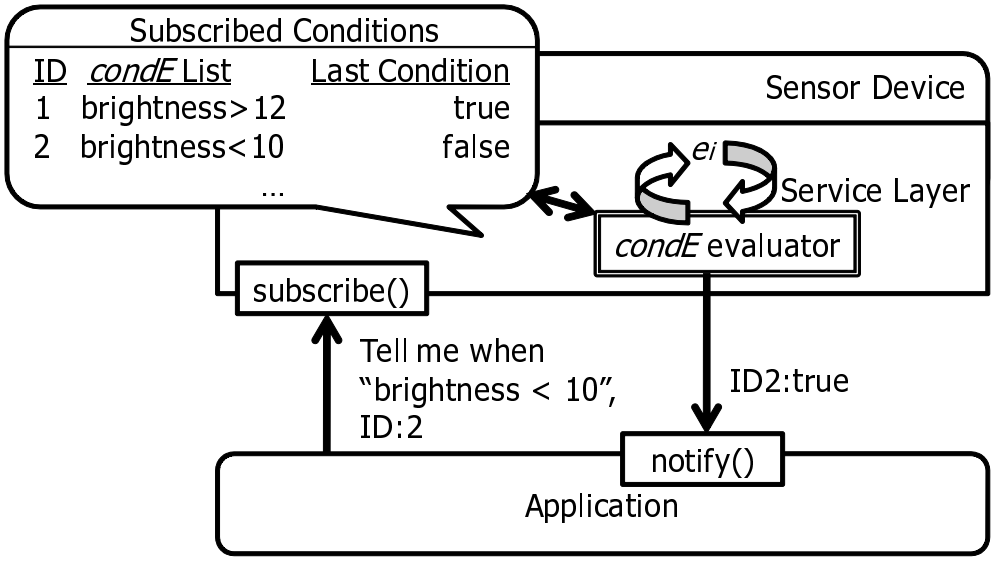

図 4 センサへの条件登録

真偽値が変化したときに通知する Web サービス [9] の WSDL [13](条件文を登録したアプリケーションの WSDL の場合と，実際に駆動させる家電機器などの WSDL の場合がある．) を登録するためのものである．サービスレイヤでは，アプリケーションから条件が登録されると，継続的に監視しているセンサデバイスから得られた環境属性値を利用して，登録された条件文の真偽値を判定する．そして，条件文の真偽値の変化に応じて条件と共に登録した登録した Web サービスを呼び出すことで通知を行う．

センサ駆動アプリケーションにおいて機器を駆動させる場合，ルールに基づいた機器の駆動パターンとして以下の 2 つの種類がある．

$$if(condition)\ then\ \{\ action\ \} \tag{8}$$

$$while(condition)\ \{\ action\ \} \tag{9}$$

条件文 (8) では，指定した条件文 $condition$ が**真になったとき**に一度だけ動作 $action$ が呼び出される．そのため，センサ駆動アプリケーションでは登録された

条件文が真になったかどうかのみを監視し，通知すればよい．それに対し式 (9) では，指定した条件文が**真の状態である間**動作 $action$ を駆動させ続ける．これは，条件が真になったときと偽になったときの 2 回，動作 $action$ を呼び出すことで機器の駆動を切り替え，条件文に従った機器の駆動を行える．すなわち，式 (9) は式 (10) のように書き換えることができる．

$$\begin{cases} if(condition)\ \{\ start\_action\ \} \\ if(!condition)\ \{\ stop\_action\ \} \end{cases} \tag{10}$$

これはすなわち，条件文の**変化があったとき**に通知を行うことで，これら 2 つの機器の駆動パターンに対応できる．そのため提案フレームワークでは，条件が真のときのみ通知を行う通常の Publish/Subscribe 型通信 [10] を改良し，条件が変化したときに通知を行うといった通信を行うこととした．

3.4 メタセンサの導入

機器の駆動条件として単一のセンサのみを用いたセンサ駆動サービスは 3.3 節で述べたセンサを用いることで実現できる．しかし，よりきめ細やかに環境情報を取得し，それに応じて動作する機器に通知を行うために，単一のセンサだけでなく複数のセンサを用いた条件文を記述することによってセンサ駆動サービスを構築する必要がある．複数のセンサを含む条件を記述する場合，その条件文を解析し，単一のセンサを用いる条件文に分解しなければならない．そのような機能を持つ解析器にセンサのサービスレイヤを付加したセンサをメタセンサと呼び，その模式図を図 5 に示す．

メタセンサは，3.3 節で示した提案センサを 2 つ以上組み合わせ，式 (7) で示す複数のセンサをまたがる条件式を解釈できるようにしたソフトウェアによるセンサである．メタセンサにはサービスレイヤに加え，複数のセンサをまたがる条件を解析する解析器 (divider) と，メタセンサに登録している単体センサから通知を受け取る $notify$ インタフェースを備える．またメタセンサは，メタセンサが利用する単体センサの一覧を保持している．

メタセンサは次の手順で動作・利用する．

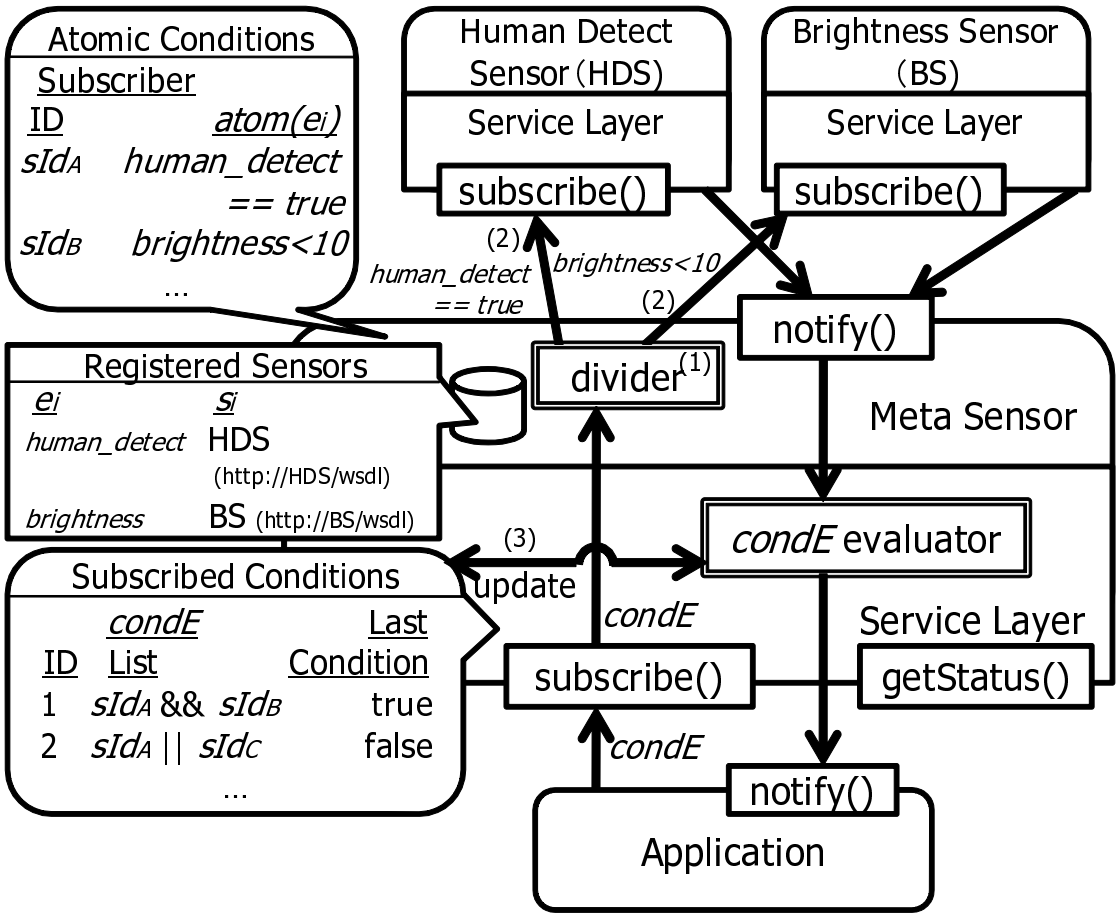
Atomic Conditions
Subscriber
ID
atom(ei)
sIdA
human_detect == true
sIdB
brightness<10
...
Human Detect Sensor (HDS)
Service Layer
subscribe()
Brightness Sensor (BS)
Service Layer
subscribe()
(2)
human_detect == true
brightness<10
(2)
notify()
Registered Sensors
ei
si
human_detect
HDS
(http://HDS/wsdl)
brightness
BS (http://BS/wsdl)
divider(1)
Meta Sensor
(3)
update
condE evaluator
Subscribed Conditions
condE List
Last Condition
ID
1
sIdA && sIdB
true
2
sIdA || sIdC
false
...
condE
Service Layer
subscribe()
getStatus()
condE
notify()
Application

図 5 メタセンサの構造

1. ユーザはまず，メタセンサに対し複数のセンサをまたがる条件文を通常のセンサと同様に $subscribe()$ メソッドを呼び出すことで登録する．

2. メタセンサは，登録された条件文を解析器 (divider) によって単一の条件文の組に分割し，それぞれの単体センサ (図 5 における Human Detect Sensor, Brightness Sensor) に対して条件文を登録する．

3. 単体のセンサがそれぞれセンサデバイスから得られたセンサ値を監視し，登録された条件文の真偽値を判定する．

4. 条件文の真偽値が変化すれば，単体センサからメタセンサに対して $notify()$ メソッドを呼び出すことにより通知 (notify) を行う．

5. メタセンサは，$notify()$ メソッドの呼び出しで送られた単体センサの真偽値を用いて，複数のセンサに対する条件文を含む論理式 $condE$ を条件の真偽を判定する $condEEvaluator$ で評価する．

6. $condEEvaluator$ で評価された結果，真偽値が変化すれば，アプリケーションにその真偽値を $notify()$ メソッドを呼び出すことにより通知する．

複数のセンサに対する駆動条件を用いたセンサ駆動サービスを構築する場合，メタセンサを用いない場合であれば，アプリケーションがそれぞれの単体センサに条件文を登録し，それぞれの単体センサから得られた条件文 $atom(e)$ の真偽値からさらに機器を駆動させるための条件文を判定する必要があった．メタセンサを用いると，複数のセンサをまたがる論理式 $condE$ の条件判定や，それぞれの単体センサへの登録をメタセンサが行うため，アプリケーションはメタセンサに対して単体センサを使う場合と同様に条件文を一度登録するだけでよい．これにより，アプリケーションでの条件文 $condE$ の解析と条件判定を行うプログラムを再開発する必要がなくなるというメリットがある．

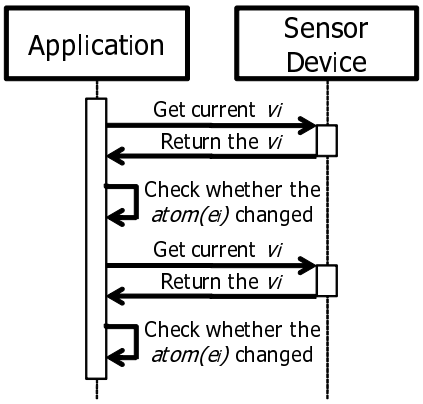

図 6 従来のセンサ駆動サービスのシーケンス図

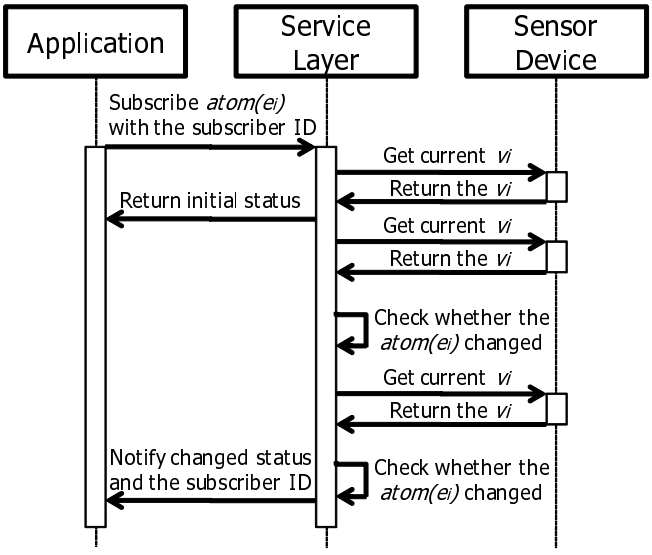

図 7 提案フレームワークのシーケンス図

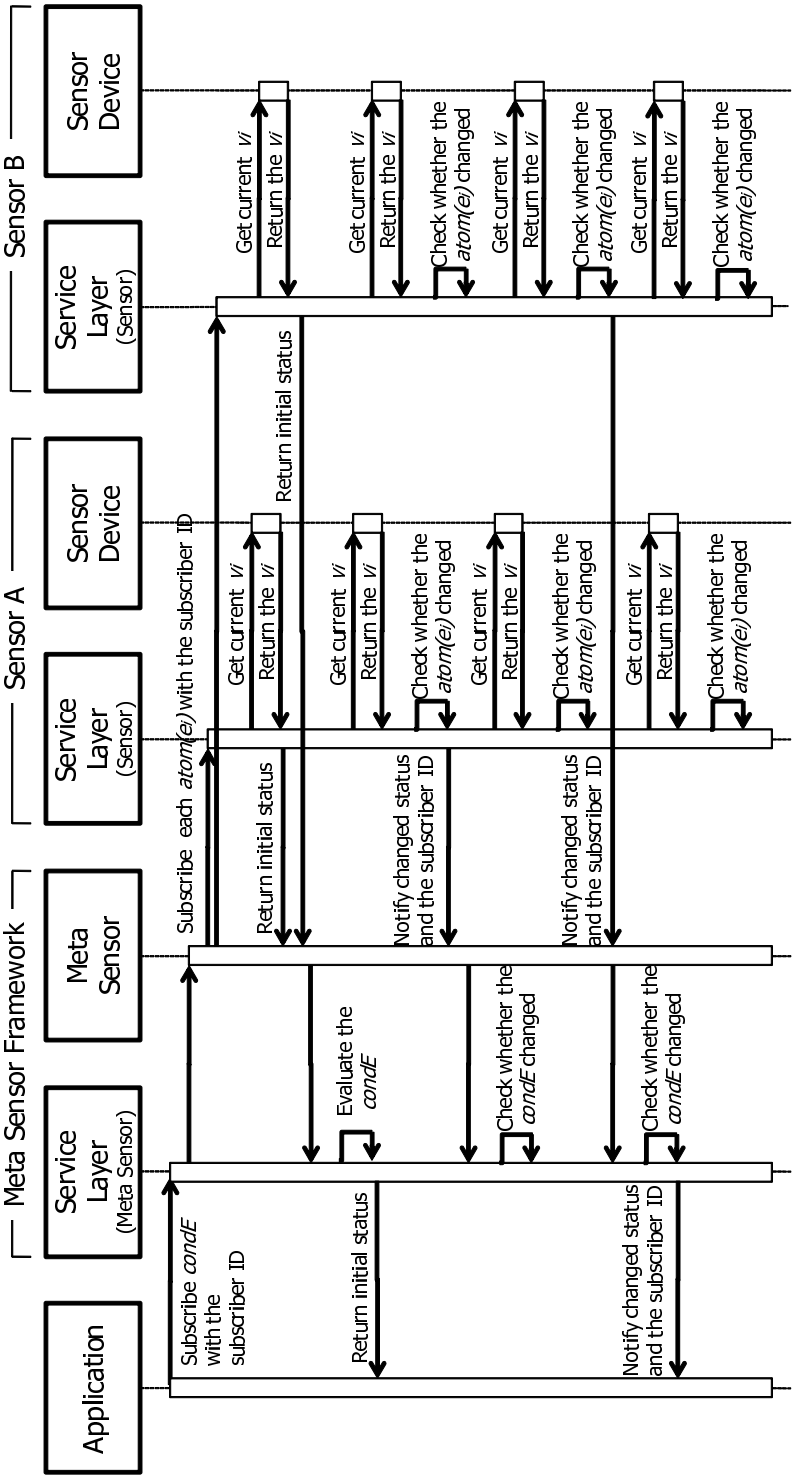

図 8　メタセンサのシーケンス図

3.5 フレームワークの動作

従来のセンサのシーケンス図 [14] を図 6 に，サービスレイヤを用いた提案フレームワークにおけるセンサのシーケンス図を図 7 に，メタセンサを用いたセンサのシーケンス図を図 8 に示す．

図 6 で示す従来のセンサでは，ネットワークを介して通信が行われるアプリケーションとセンサ間で頻繁にメッセージ交換が行われている．開発者が決めたセンサ値取得の間隔ごとに必ず通信が発生してしまう．またアプリケーションは，センサ値 v_i，値変換関数 f_i などのセンサ固有の情報も把握しないといけなかった．

図 7 に示す提案フレームワークでは，ネットワークを介して行われる通信は条件を登録する時，真偽値の変化を返す時の 2 通りだけであり，条件が変化しなければ通信は発生しない．サービスレイヤとセンサデバイス間の通信はネットワークを介したものではないため，ネットワークには負荷をかけない．アプリケーションとサービスレイヤ間は登録した条件式の真偽値が変化したという必要最低限の通信となっている．

同様に，メタセンサを用いた場合でも，アプリケーションとメタセンサ間，メタセンサと単体センサ間で通信が必要最低限となっている．

次に，2.5 節で述べた問題点について考える．

問題 1: センサとアプリケーションの結合が密

センサとアプリケーション間の通信は，条件の登録とその条件文の変化の通知のみである．アプリケーションはセンサについて次の 3 つの情報のみを知っておけばセンサを利用することが出来る．

- センサ s_i の WSDL
- センサ s_i のプロパティ p_i
- プロパティ p_i の型制約 t_e

これは，センサ固有のセンサ値 v_i や変換関数 f_i，センサ値の型制約 t_v を知らなくても利用できることを意味し，センサとアプリケーション間の結合が疎になっていることを示している．

問題 2: スケーラビリティが低い

スケーラビリティを考慮する上で問題となってくるのが，センサデバイスやそれを利用するアプリケーションが増加することによるアプリケーションとネットワークの負荷である．アプリケーションの負荷については，値変換関数による値変換の処理がセンサのサービスレイヤに委譲しているため，アプリケーションでの値変換処理をする必要がない．しかし，サービスレイヤでその処理を行わなければならないため，総合的には変化していない．それに対し，ネットワークの負荷については，図 6 でセンサ値を一定時間おきに取得していた状態から，条件登録時と条件文の真偽値が変化したときの 2 パターンに減少している．

これらのことから，提案フレームワークは 2.5 節で述べた問題点 1，問題点 2 を解決する設計になっているといえる．

3.6 実装

前節までで提案したフレームワークを Java 言語で実装し，設計の実現性を確かめた．実装したフレームワークのクラス図を図 9 に示す．

図 9 において，センサは抽象クラスである *AbstractSensor* クラスを継承した実装クラスと *ServiceLayer* クラスの 2 つからなり，メタセンサも *AbstractSensor* クラスを継承した実装クラスとして実装する．

ServiceLayer クラスはどの単体センサ，メタセンサにおいても共通で利用される *subscribe* メソッドや *subscribe* された条件式の解釈・条件判定を行う *run* メソッドを持つ．センサで行うべき処理のうち，このような共通機能の部分を *ServiceLayer* クラスとしてまとめることで，それらの機能の再開発コストを抑えることが出来る．

センサデバイスは *AbstractSensor* クラスの継承クラスとし，実装すべきメソッドを *AbstractSensor* クラスで定義した．また，センサごとに用意される *Sensor* クラス (図 9 では，例として *SingleSensor* クラスで示している) では *ServiceLayer*

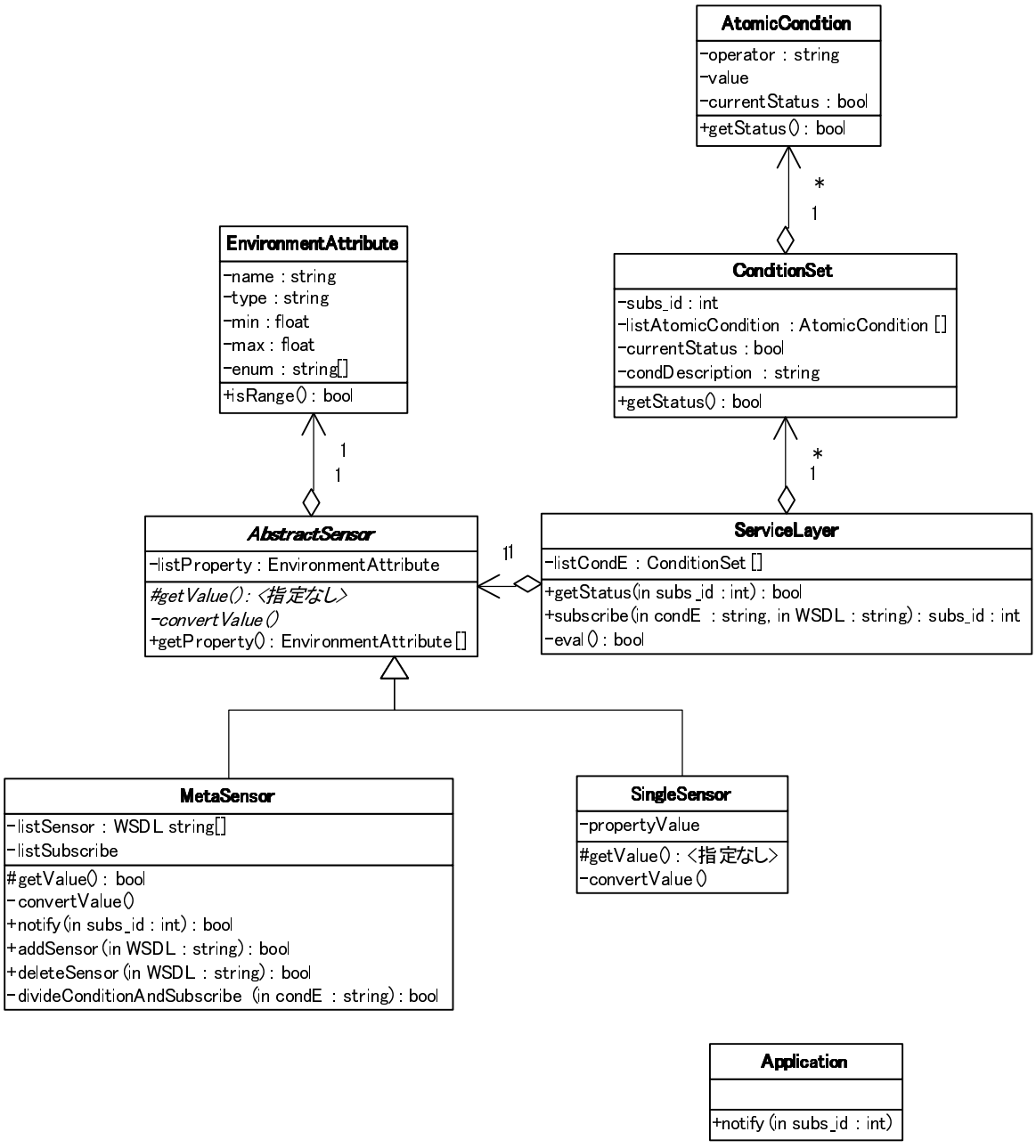

図 9 実装した提案フレームワークのクラス図

クラスで扱わないセンサ固有の値や処理，すなわちセンサデバイスからの値取得メソッドや値変換関数 f を実装することとした．

メタセンサは，単一のセンサと同様に *ServiceLayer* クラスに接続して扱えるようにするため，*AbstractSensor* クラスの継承クラスとした．*MetaSensor* クラスでは，あらかじめメタセンサをなすセンサの登録や削除，メタセンサの条件文を登録するためのメソッド *addSubscribe*，*removeSubscribe* 等を追加した．

登録された条件文を保持し，与えられたセンサ値から真偽を判定するクラスとして *ConditionSet* クラスを定義した．*ServiceLayer* クラスの *subscribe* メソッドにより登録された条件文をパースすることによりインスタンスが生成される．*ConditionSet* クラスは複数の *AtomicCondition* クラスや *CondTree* クラスを保持し，木構造で連結されている．

AbstractSensor クラスは，センサがとりうる環境属性値 e の型制約 t_e を判定する *EnvironmentAttribute* クラスを持ち，アプリケーションからのリクエストに応じて *EnvironmentAttribute* クラスのインスタンスを返す．アプリケーションはセンサから得られたその *EnvironmentAttribute* クラスの情報を用いて型制約 t_e を知ることができる．

次章では，実装したこのフレームワークを用いて実際にセンサ駆動サービスを動作させ，アプリケーションやネットワークの負荷を測定した．

4. 評価実験

4.1 評価の概要と目的

本章では，従来利用されてきたセンサ駆動アプリケーションに代わり本論文で提案するセンサ駆動アプリケーションを利用した場合，従来法と比較してアプリケーションにおける条件判定の負荷，ネットワークを利用することによるトラフィック負荷がどの程度増大・減少するか，2.5 節で提示した問題点をどの程度解決できているかを実際に提案フレームワークを実装し，動作させることで実験的に評価する．

4.2 実験環境

提案するフレームワークを実際に実装し，センサとセンサアプリケーション間の通信がどの程度あるかを実測した．実験環境における実験機の構成を図 10 に示す．また，実験に利用した実験機環境のハードウェアを表 5 に，実験機のソフトウェアの環境を表 6 に，実験で使用した Phidgets の力センサと明るさセンサの仕様を表 7 に，それらの写真を図 11，図 12 に示した．

図 10 に示すとおり，コンピュータ A に Phidgets の力センサを，コンピュータ B に明るさセンサを設置した．それぞれのセンサは提案フレームワークにおける *subscribe* メソッドや条件判定機構を持ち，各コンピュータで公開されている Web サービスのインタフェースを経由してそれらのメソッドにアクセスし，条件を登録することができるようになっている．コンピュータでは 0.1 秒おきにセンサ値を取得し，変換関数を用いて環境属性値に変換し，登録された条件の真偽を判定している．センサ値が変化することで条件の真偽値が変化すれば，条件を登録したクライアントに対して notify メソッドを呼び出すようになっている．

また図 10 においてコンピュータ C にメタセンサを設置した．このメタセンサは力センサと明るさセンサを組み合わせたメタセンサで，2 つのセンサの値を組み合わせた条件文を登録することができる．

この環境において，2 つの動作パターンで動作遅延を測定した．まず従来法の

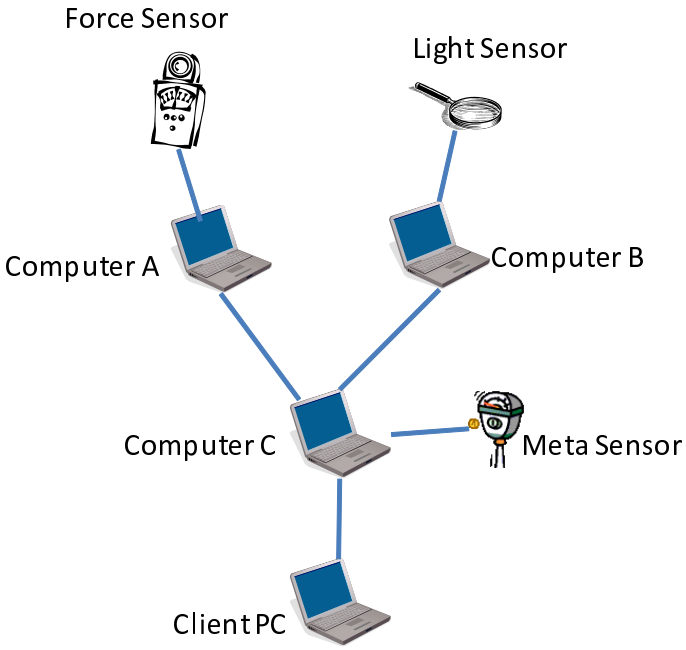

図 10　実験機の構成

表 5　実験機環境

	コンピュータ A	コンピュータ B
CPU	Core Duo U2400 1.06GHz	PentiumIII 750MHz
メモリ	2048MB	512MB
OS	WindowsVista Business	WindowsXP SP2
ネットワーク	100BASE-TX	100BASE-TX
	コンピュータ C	クライアント PC
CPU	PentiumM 1GHz	Core Duo U2400 1.06GHz
メモリ	768MB	2048MB
OS	WindowsXP SP1	WindowsVista Business
ネットワーク	100BASE-TX	100BASE-TX

表 6 実験環境 (ソフトウェア)

Java	Java 2 Standard Edition (build 1.5.0_11-b03)
Web サーバ	Tomcat 5.5
Web サービスエンジン	Axis2[15]

表 7 Phidgets センサの仕様

センサ名	プロパティ p	型制約 t_v	変換関数 f
力センサ	$ForceSensor$	$int\{0, 1000\}$	$e = \begin{cases} true & \text{for } v \geq 500 \\ false & \text{for } v < 500 \end{cases}$
明るさセンサ	$LightSensor$	$int\{0, 1000\}$	$e = v$

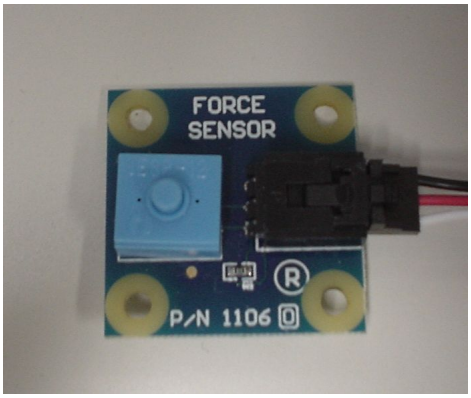

図 11 Phidgets 力センサ

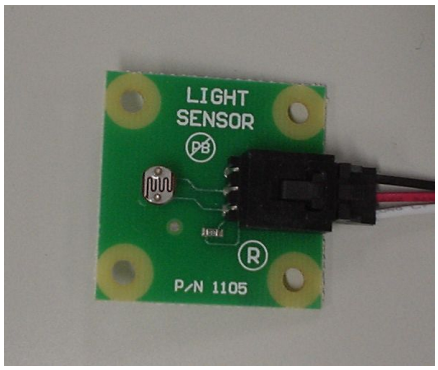

図 12 Phidgets ライトセンサ

動作パターンとして，クライアント PC で動作するクライアントアプリケーションからセンサの Web サービスに対して一定時間おきにセンサ値を取得し，クライアントアプリケーションにおいてセンサ値を環境属性値に変換し，条件判定を行い，機器を駆動させた．また，提案法の動作パターンとして，条件文をメタセンサに登録することで，メタセンサが必要な個別の条件文を力センサと明るさセンサに対して登録し，条件の変化を通知してもらうこととした．

4.3 実験手順

4.2 節で示す実験環境において，次の実験手順で実験を行った．

手順 1. タイムスタンプ出力のためのプログラムコードの挿入

まず，実装したフレームワーク中で時間を測定したいポイントに，現在の時間を出力するコードを挿入した．タイムスタンプ出力のためのコードは，Java 言語の *java.util.logging* パッケージを用い，ミリ秒単位で出力することとした．出力される時間の誤差はコンピュータに搭載されているオペレーティングシステムの時刻精度に依存し，今回実験で用いている Windows シリーズではおよそ 10 ミリ秒である．

手順 2. コンピュータの時刻の同期

コンピュータ間のネットワーク負荷を測定するため，各コンピュータの時刻を出来る限り同一にしておく必要がある．まず，それぞれのコンピュータに対して，同じネットワークに設置してある Network Time Protocol(NTP) サーバに接続し，コンピュータの時間を同期させる NTP[16] を用いて時刻を同期させた．各コンピュータの時間のずれは，30 ミリ秒以内となった．手順 1 で述べたオペレーティングシステムの時間誤差とあわせて，本実験における時間誤差は 40 ミリ秒程度である．

手順 3. 条件の登録

次に，クライアントアプリケーションから力センサと明るさセンサの二つのセンサの値を条件式に持つ以下のような条件式をメタセンサに対して登録した．

$$ForceSensor\ eq\ false\ \&\&\ LightSensor\ <\ 50\ then \\ ::\ http://163.221.172.122:8080/axis2/services/SAClientService/\ ::\ notify \tag{11}$$

条件式 (11) では，$ForceSensor$ が押されていない状態，かつ $LightSensor$ で取得した明るさが 50 ルクス未満となったとき，条件を登録したクライアントアプリケーションに対して $notify$ メソッドを呼び出すといったことを表している．

従来手法では，上記の条件文に該当する条件判定のプログラム (図 13) を Java 言語で実装し，動作させた．(なお，分かりやすさのため，一部プログラムを書き換えている．)

手順 4. センサの監視と条件判定・通知

条件が登録されると，それぞれのセンサは自動的にセンサの値を監視し，取得したセンサの環境属性値 e を用いて登録されている条件式を判定する．状態が偽から真に変化すれば $true$ を，真から偽に変化すれば $false$ を $notify$ メソッドを呼び出すことでクライアントアプリケーションに通知する．

```
if(ForceSensor.status == false &&
        LightSensor.currentValue < 50) {
    System.out.println("invoke notify method.");
}
```

図 13 実験で使用した条件文の Java 言語による実装

従来手法では，それぞれのセンサの $getValue$ メソッドを 100 ミリ秒おきに呼び出し，現在のセンサ値を取得し，変換関数を用いて環境属性値に変換し，図 13 のコードで条件判定した．

これらの動作のいくつかに時間を測定するポイントを設け，タイムスタンプを記録していく．この試行を 10 回繰り返し，10 回の平均を結果として用いる．

これらの手順に加え，センサ値の変化により $notify$ メッセージを送信させるため，表 8 に示す順番でセンサ周辺の環境を変化させた．力センサにおいて，力センサの状態が off であるとは，力センサのボタンを押していない状態，on は押している状態，明るさセンサにおいて，$bright$ はセンサを手で覆わず明るい状態，$dark$ はセンサを手で覆い，暗い状態を示す．$Phase0$ はメタセンサからの $subscribe$ が来た直後の状態，$Phase1$ は力センサを押した状態，$Phase2$ はそれに加え，明るさセンサを暗くした状態といったそれぞれのセンサの状態を表す．

この手順に従って，提案フレームワークによるセンサを用いた場合と，従来型のセンサを用いた場合の 2 通りで動作させ，それらを比較することでアプリケーションとネットワークの負荷を測定した．

4.4 結果

従来法における実験の測定結果を表 9 に，提案法の測定結果を表 11 に示す．

なお，表 9 において，従来法における値取得とその判定処理はセンサ駆動サービスの駆動条件の結果によらず毎回行われるため，$Phase$ による区別を行ってい

表 8 実験フェーズ

	力センサの状態	力センサの条件文の真偽	明るさセンサの状態	明るさセンサの条件文の真偽	メタセンサの条件文の真偽
Phase 0	off	true	bright	false	false
Phase 1	on	false	bright	false	false
Phase 2	on	false	dark	true	false
Phase 3	off	true	dark	true	true
Phase 4	off	true	bright	false	false

表 9 結果 (従来法)

no.	測定場所	測定要素	平均時間 (msec)
1	クライアント→力センサ	通信	3605.00
2	力センサ	処理	137.31
3	力センサ→クライアント	通信	614.00
4	クライアント	処理	0.00
5	クライアント→明るさセンサ	通信	5341.00
6	明るさセンサ	処理	617.17
7	明るさセンサ→クライアント	通信	−959.00

ない．また，表 11 における測定要素の項目で,「処理」は同一 PC 内での条件判定などの処理を指し,「通信」は PC から PC への Web サービス呼び出しを含む通信速度を測定したことを示している．

各表において測定した時間に負の値が含まれる．これは，測定に利用したコンピュータ間の時間のずれのために発生する．この時間のずれを排除するため，コンピュータ間の通信における行きと帰りの合計をとり，その結果を表 10，表 12 に示す．

表 10 従来法における速度の平均時間

no.	測定場所	測定要素	平均時間 (msec)
1	力センサ	処理	137.31
2	明るさセンサ	処理	617.17
3	クライアント⇔力センサ (往復)	通信	4219.00
4	クライアント⇔明るさセンサ (往復)	通信	4382.00

5. 考察

5.1 フレームワークの特徴

従来法と比較した提案フレームワークの動作を，以下の観点から考察を行った．

センサとアプリケーションの疎結合性

まず，2.5 節で述べた問題点について考察する．問題点 1 として，センサとアプリケーションの密結合の問題があった．従来では，アプリケーションがセンサを扱う場合，センサの詳細な仕様を知っておく必要があった．しかし，提案フレームワークを用いることで，センサにおけるセンサ値の型制約や，センサ値を環境属性値に変換する変換関数を知っておく必要がなくなった．実験で用いた条件式である式 (11) においても，提案フレームワークではセンサ固有のセンサ値や変換関数をセンサから通知を受けるアプリケーションに記述していない．このことから，センサとアプリケーション間で疎結合性が高まったと言える．それにより，実験で使用した Phidgets の力センサや明るさセンサを別ベンダ製の力センサ，明るさセンサに置き換えても，アプリケーションにはほとんど変更を加えずに同様に利用することができる．

上記の点から，2.5 節におけるセンサ駆動サービスの問題点 1 は解決できたといえる．

スケーラビリティ

表 11 結果 (提案法)

no.	フェーズ	測定場所	測定要素	平均時間 (msec)
1	subscribe	クライアント→メタセンサ	通信	4896.00
2		メタセンサ	処理	89.23
3		メタセンサ→力センサ	通信	397.60
4		メタセンサ→明るさセンサ	通信	938.00
5		力センサ	処理	81.00
6		明るさセンサ	処理	257.00
7		力センサ→メタセンサ	通信	73.70
8		明るさセンサ→メタセンサ	通信	283.80
9		メタセンサ	処理	9.36
10		メタセンサ→クライアント	通信	−3.70
11	Phase1	力センサ→メタセンサ	通信	52.40
12		メタセンサ	処理	5.73
13		メタセンサ→力センサ	通信	35.70
14	Phase2	力センサ→メタセンサ	通信	61.90
15		メタセンサ	処理	5.38
16		メタセンサ→力センサ	通信	−9.70
17	Phase3	明るさセンサ→メタセンサ	通信	604.00
18		メタセンサ	処理	2.74
19		メタセンサ→明るさセンサ	通信	872.00
20	Phase4	力センサ→メタセンサ	通信	60.40
21		メタセンサ	処理	69.14
22		メタセンサ→クライアント	通信	−4.00
23		クライアント→メタセンサ	通信	38.00
24		メタセンサ→力センサ	通信	−10.20
25	Phase5	明るさセンサ→メタセンサ	通信	960.50
26		メタセンサ	処理	44.43
27		メタセンサ→クライアント	通信	−4.40
28		クライアント→メタセンサ	通信	36.40
29		メタセンサ→明るさセンサ	通信	−8.10

表 12 提案法における速度の平均時間

no.	フェーズ	測定場所	測定要素	平均時間 (msec)
1	条件登録時	クライアント⇔メタセンサ	通信	4892.30
2		メタセンサ⇔力センサ	通信	471.30
3		メタセンサ⇔明るさセンサ	通信	1221.80
4		メタセンサ (合計)	処理	98.59
5		力センサ	処理	81.00
6		明るさセンサ	処理	257.00
7	監視時	力センサ⇔メタセンサ	通信	63.83
8		明るさセンサ⇔メタセンサ	通信	1214.20
9		メタセンサ⇔クライアント	通信	33.00
10	監視時 (notify なし)	メタセンサ	処理	4.46
11	監視時 (notify あり)	メタセンサ	処理	56.79

次に，問題点 2 であるスケーラビリティの問題を考える．実験ではセンサを 2 個，メタセンサを 1 個使用したが，さらにセンサ数を増加させた場合，従来法では表 10 における No.1〜No.4 の値を合計した処理時間が毎センサアクセスごとにかかってしまうことに加え，センサアクセスは 100 ミリ秒おきなどの短い時間おきに通信が発生する．そのため，同一ネットワーク上で利用できるセンサの数には限界がある．一方で，提案手法を用いた場合，条件登録時に表 12 における No.1〜No.6 の処理時間の合計がかかり，その後は No.7,8,10 を合計した処理時間，状態によっては No.7,8,9,11 を合計した処理時間がかかる．いずれの場合においても，従来法と比較してかかる時間が少ない．さらに，提案法では従来法のように絶えずネットワークを経由して値取得のためのアクセスを行うのではなく，登録した条件文の真偽値が変化したときのみ通信を行うため，同一ネットワークに接続されているセンサ数は従来法と比較して増やすことができる．また，センサ値は絶えず変化するが，登録されているセンサ駆動の条件文は頻繁に状態が変化することは少ないため (例えば，今回の実験においては，1 回の実験あたりセンサ値は数百回変化したが，条件文の真偽値の変化による $notify$ メ

ソッド呼び出しの通知は5回であった)，さらに通信コストは抑えられる．

これらのことから，センサ駆動サービスの問題点2が解決できたといえる．

ネットワークの負荷

従来法では，クライアントアプリケーションとセンサ間の通信が，表10より力センサの場合で約140ミリ秒，明るさセンサの場合で約620ミリ秒あった．提案法では，表12より条件登録時に力センサで約80ミリ秒，明るさセンサで約260ミリ秒，センサ監視時に力センサで約60ミリ秒，明るさセンサで約1210ミリ秒であった．

従来法では，一定時間おきにネットワークを介してセンサ値を得るための通信が行われるため，上記の負荷が常にかかってしまう．一方で，提案法を用いた場合では，条件が変化したときのみ，上記の負荷がかかる．また，その負荷も従来法におけるセンサ値を得るための通信における負荷程度であるため，大きな負担とはならないと言える．

このようなネットワーク負荷の減少は，インターネットを経由したセンサの状態を監視したい場合において効果を発揮する．アプリケーションがインターネットを経由してセンサの情報を得たい場合，従来の方法ではトラフィック量が多くなりすぎるため，利用できなかった．しかし，提案手法を用いることで，ネットワークのトラフィック量が削減され，利用できるようになる．それにより，インターネット上にさまざまなセンサを公開し，ユーザは利用したいセンサを自由に利用するといったことも実現可能となる．

5.2 フレームワークの限界

センサ駆動サービスにおいてセンサやそれを利用するアプリケーションが増加した場合に発生する問題点を，提案フレームワークを用いることで解決できた．しかし，条件判定処理のセンサへの委譲によって，アプリケーション側での処理は減少するが，逆にセンサでの処理量が増加してしまうといった別の問題が発生する．すなわち，センサでは登録された条件文を解釈し，その条件文をメモリに

保存し，一定時間おきに取得したセンサ値を用いてそれらの条件文の真偽値を判定しなければならない．そのため，従来よりもセンサに持たせるハードウェアの性能が求められる．しかし，センサ数やセンサを利用するアプリケーション数が多い場合には，提案フレームワークを用いなければネットワークが疲弊してしまい，センサネットワークを複数設置しなければ運用することが出来なくなる．その点を考慮すると，センサ数やアプリケーション数が多い場合，それらが増加すればするほど提案手法を用いることでコストの問題は極小となる．

また，提案フレームワークはリアルタイムにセンサの値を監視したいようなアプリケーションには用いることができない．なぜなら，利用される状況として，センサの値が一定の条件を満たしたときに何らかのアクションを行うという利用形態を想定しており，アプリケーションには条件の真偽値が変化したときしか現在の値を知る方法がないからである．このような用途のアプリケーションのために，従来のセンサ値を一定時間おきに取得するためのメソッド $getStatus$ を互換性のために提供している．そのため，一定時間ごとにセンサに値を取得しに行くという従来の方式を併用することができる．しかし，その方法をとった場合，リアルタイム性を求めるアプリケーションにおいては 2.5 節で述べた問題点のいずれも解決できなくなる．

5.3 先行研究との比較

幸島ら [17] は，複数の異なるセンサ情報を統一的に管理し，高次のサービスと連携させる機能を提供するミドルウェアである SENSORD を提案している．SENSORD では，センサデバイスにセンササーバを付属させ，センサデバイスから得られたデータをネットワークを介して SENSORD ミドルウェアに蓄積する．ユーザは SENSORD スクリプトを用いてルールを記述しておき，蓄積されたセンサデータに対して評価を行い，ルールが適合する場合に対応するサービスを起動する．

本論文における提案フレームワークと SENSORD との違いは，ユーザが登録したセンサ駆動サービスのルールをセンサが評価するか，ミドルウェアでセンサデータを蓄積しておき，SENSORD アプリケーションで評価するかの違いである．

SENSORD では，ネットワークを介して SENSORD アプリケーションにセンサデータがそのまま保存される．しかし，センサデータがネットワークを流れるため，2.5 節で提示したスケーラビリティの問題の解決にはなっていない．ただし，SENSORD ではセンサデータを蓄積しているため，過去のデータを利用するようなスクリプトを記述できることが SENSORD のアドバンテージとなっていると考えられる．

6. おわりに

本論文では，センサの情報を元に機器を駆動させるセンサ駆動アプリケーションにおいて，スケーラビリティを考慮したときに発生する問題点を解決したフレームワークを提案した．

従来のセンサ駆動サービスでは，センサを利用するアプリケーションがセンサを利用する際に，センサがとる値の意味やその値を知っていなければ利用できず，センサとアプリケーションの結合が密であるという問題があった．提案したフレームワークでは，センサに標準的なインタフェースをもたせ，センサ固有の値や処理をセンサ側で処理させることで，どのセンサを利用する場合でも共通の方法で利用することが出来るようになった．

また，センサ数を増加させることによるスケーラビリティの問題に対しては，センサ駆動サービスの条件判定処理をアプリケーションからセンサ側に委譲することにより，アプリケーションは登録した条件の真偽値の変化のみを通知してもらうため，アプリケーションやネットワークにかかる負荷を軽減し，スケーラビリティを向上させた．

この提案フレームワークを実装し，アプリケーションとネットワークの負荷を測定する実験を行った．実験では，1 回あたりのネットワークの負荷は同程度ながらも，その通信回数が大きく減少させることができたため，総量を大幅に減少させることができた．これにより，従来法よりも上記 2 点の問題点に対して優位性があることを確認した．

謝辞

本研究を進めるにあたり，多くの方々に御指導，御支援を賜りました．心から深く感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 松本 健一 教授には，2 年間にわたって研究する場や機会を賜りました．また，本研究を進めるにあたっては，多大な御指導や御配慮を頂きました．心より深く感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 伊藤 実 教授には，副指導教官を担当して頂き，研究の内容について鋭い御指摘や御助言を頂きました．心より深く感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 門田 暁人 准教授には，研究全般を通して研究の地を固めていくことについて，多大な御指導，御助言を頂きました．心より深く感謝申し上げます．

神戸大学 大学院工学研究科 中村 匡秀 准教授には，研究の根幹に関わる部分から研究の進め方について多大な御指導，御助言を頂きました．また，研究論文の書き方やプレゼンテーションの作法においても，有益な数多くの知識，技術について全面的に御指導頂きました．心より深く感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 飯田 元 教授には，本研究を進めるにあたって鋭い御指摘を頂きました．心より深く感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 大平 雅雄 助教には，ソシオメトリクス班にて複雑な事象を組み立てていくという論理的なものの考え方について丁寧な御指導を頂きました．心より深く感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 川口 真司 助教には，共起関係分析班にて鋭いご指摘や，研究の進め方について御指導を頂きました．心より深く感謝申し上げます．

神戸大学 大学院工学研究科 井垣 宏 助教には，本フレームワークの実装における技術的な御支援にはじまり，研究の内容や進め方について鋭い御指摘，御指導を頂きました．また，論文の書き方やプレゼンテーションの作成においても有益な御助言，御助力を頂きました．心より深く感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 森崎 修司 助教には，本研究を

進めるにあたって鋭い御指摘を頂きました．心より深く感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 博士後期課程 山内 寛己 氏には，本論文の作成にあたって多大な御指導，御指摘を頂きました．心より深く感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 Web サービス班の皆様には，週次のミーティングにおいてさまざまな御意見や御助言を頂きました．心より深く感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア工学講座 およびソフトウェア設計学講座の皆様には，日頃の研究生活においてさまざまな御助言を頂き，心地よい研究環境を与えて頂きました．心より深く感謝申し上げます．

最後に，2 年間の大学院での研究生活において，陰ながら支えてくれた祖母，父，母，妹に心より深く感謝いたします．

参考文献

[1] H. Igaki, M. Nakamura, and K.-I. Matsumoto. Design and Evaluation of the Home Network Systems Using the Service Oriented Architecture. In *Proceedings of the International Conference on E-Business and Telecommunication Networks(ICETE'04)*, Vol. 1, pp. 62–69, August 2004.

[2] M. Nakamura, A. Tanaka, H. Igaki, H. Tamada, and K.-I. Matsumoto. Adapting Legacy Home Appliances to Home Network Systems Using Web Services. In *IEEE International Conference on Web Services (ICWS'06)*, pp. 849–858, September 2006.

[3] 田中章弘, 中村匡秀, 井垣宏, 松本健一. Web サービスを用いた従来家電のホームネットワークへの適応. 電子情報通信学会技術研究報告. Vol. 105, No. 628, pp. 067–072, March 2006.

[4] Matsushita Electric Works Ltd. Katteni switch. http://biz.national.jp/Ebox/katte_sw/.

[5] Daikin Industries Ltd. Air conditioner. http://www.daikin.com/global_ac/.

[6] Nabtesco Corp. Automatic entrance system. http://nabco.nabtesco.com/english/door_index.asp.

[7] TOTO USA Inc. Sensor Faucet. http://www.totousa.com/prodcatalog.asp?cid=54.

[8] J. Nehmer, M. Becker, A. Karshmer, and R. Lamm. Living Assistance Systems: An Ambient Intelligence Approach. In *Proceedings of the 28th International Conference on Software Engineering(ICSE'06)*, pp. 43–50, 2006.

[9] World Wide Web Consortium. Web Services Activity. http://www.w3.org/2002/ws/.

[10] G. Mühl. Generic Constraints for Content-Based Publish/Subscribe. In *Proceedings of the 9th Internationall Conference on Cooperative Information Systems(CooplS'01)*, pp. 211–225, 2001.

[11] S. Greenberg and C. Fitchett. Phidgets: Easy Development of Physical Interfaces through Physical Widgets. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology(UIST'01)*, pp. 209–218, 2001.

[12] Network Working Group. Augmented BNF for Syntax Specifications: ABNF. http://www.ietf.org/rfc/rfc4234.txt.

[13] World Wide Web Consortium. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[14] Object Management Group Inc. Unified Modeling Language (UML), version 2.1.1. http://www.omg.org/technology/documents/formal/uml.htm.

[15] Apache Software Foundation. Apache Axis2. http://ws.apache.org/axis2/.

[16] Network Working Group. RFC:1305 Network Time Protocol(Version 3). http://www.ietf.org/rfc/rfc1305.txt.

[17] A. Sashima, Y. Inoue, and K. Kurumatani. Spatio-Temporal Sensor Data Management for Context-Aware Services: Designing Sensor-Event Driven Service Coordination Middleware. In *Proceedings of the 1st International Workshop on Advanced Data Processing in Ubiquitous Computing (ADPUC'06)*, 2006.

付録

A. BNF 記法によるルール記述

```
ConditionSetExpression::
    ConditionalOrExpression "then" AtomicAction ( ',' AtomicAction )*

ConditionalOrExpression::
    ConditionalAndExpression ( "||" ConditionalAndExpression )*

ConditionalAndExpression::
    AtomicConditionExpression ( "&&" AtomicConditionExpression )*

AtomicConditionExpression::
    AtomicCondition
    AtomicConditionWithNot

AtomicConditionWithNot::
    "!" AtomicCondition

AtomicCondition::
    SensorName Operator CompareValue

SensorName::
    id

Operator::
    "<"
    ">"
    "<="
```

```
    ">="
    "=="
    "!="
    "eq"

CompareValue::
    id
    TrueFalse

TrueFalse::
    "true"
    "false"
```

B. センサ駆動アプリケーションのメソッドとその引数

センサ駆動サービスを構成する各モジュールが実装すべきメソッドと，とるべき引数をまとめた．なお，実装は Java 言語で行ったため，一部 Java 言語に依存する制約が含まれる．

B.1 クライアントアプリケーション

クライアントアプリケーションとは，センサに対して条件文を登録し，条件が変化したときに *notify* を受けるアプリケーションである．以下に，メソッド一覧とその引数を示す．

notify

センサ駆動サービスに対して登録した条件文の真偽値が変化したときに送られる通知は，このメソッド呼び出すことで行われる．

int sensorId センサ固有の ID

int subscribeId 登録時に戻り値として得られる登録条件 ID

boolean status 最新のステータス

B.2 サービスレイヤ

サービスレイヤとは，センサ駆動サービスにおけるセンサの共通機能を実装したクラスである．以下のメソッドを持つ．

ServiceLayer

ServiceLayer クラスのコンストラクタである．

int sensorType 単体センサモードとして動作するか，メタセンサとして動作するかを切り替えるオプションである．3.6 節で示した実装では，1 を指定すると単体センサ，2 を指定するとメタセンサとして動作する．

AbstractSensor sensor センサデバイスから値を取得するセンサクラスのインスタンス

getStatus

現在の登録条件の真偽値を返す．

int subscribeId 登録条件 ID

subscribe

通知条件を登録する．

String condE 条件文

run

センサデバイスからセンサ値を取得し，環境属性地に変換し，条件文の真偽値を判定し，真偽値が変化すれば通知を行うといった，一連のセンサ監視動作を行う．一定時間おきにこのメソッドを呼び出すことで，センサ値を監視し，通知を行うことができる．引数はとらない．

notifyClient

条件文の真偽値が変化したときに，クライアントアプリケーションに通知を行うメソッドである．本メソッドのスコープはクラス内からのアクセスのみである．

ConditionSet cs 条件文が保存されているオブジェクト

getValue

センサ値を得るメソッドである．提案フレームワークのインタフェースのみでなく，従来のセンサとしても利用できるよう，本メソッドを残している．引数はとらない．

B.3 センサ

subscribe

条件を登録するメソッドである．

String condE 登録する条件文

getProperty

センサの型制約情報を示す *EnvironmentAttribute* クラスのオブジェクトを返す．引数はとらない．

B.4 メタセンサ

メタセンサは，*AbstractSensor* クラスを継承したクラスで実装する．インタフェース自体は，単体センサのものとクライアントアプリケーションのものを両方共存させたものとなっている．

B.5 ConditionSet

ConditionSet クラスは，登録された条件文が木構造で保持されたオブジェクトである．

getStatus

条件文の現在のステータスを返す．引数はとらない．

updateStatus

引数に最新のセンサ値を与えることで，保持している条件文の真偽値を再度判定し，その真偽値を返す．

Obejct o 最新のセンサ値．*int* であるか *String* であるかはセンサの型制約の違いによるので，ここでは Object 型を用いている．

B.6 AtomicCondition

ConditionSet と同様のインタフェースを持つ．それに加え，以下のコンストラクタを持つ．

AtomicCondition

AtomicCondition クラスのコンストラクタである．

int sensorType センサのタイプ

AbstractSensor sensor センサオブジェクト

int subscribeId 登録 ID

String property プロパティ名

String operator 演算子

Object value 比較する値