ネットワーク上の仮想マルチコア計算機構成法

前田 賢一[1,a)]

概要：Internet of Things（IoT）が注目される中，今後予想される多くのアプリケーションをいかに容易に作ることができるかということが，喫緊の課題である．本稿では，ネットワーク上に仮想的なマルチコア計算機を構成する方法に関して述べる．提案方法は，ネットワーク共有メモリと，ネットワーク透過なスレッドを利用するものである．これによって，ネットワーク上のプログラミングを単一計算機と同一にすることが可能となる．

How to Build a Virtual Multi-Core Computer on the Network

Ken-ichi Maeda[1,a)]

1. はじめに

近年，Internet of Things（IoT）が注目を集めており，主に応用面を中心に様々な議論がある．IoT では，従来以上に数多くの応用が存在すると期待されている．ここで重要なのは，数多くのアプリケーションをいかに容易に開発できるようにするかということである．

コンピューティング環境は，スタンドアローンから始まり，それをネットワークで結合し，さらにクラウドに処理を集中する方向に進化してきた．いつの時代にも，センターと端末という概念があり，それらを結ぶ通信線は安定していることが前提となっていた．センターと端末という概念は，システムの非対称性を生むことになり，センターにおけるプログラミングと端末におけるプログラミングとの差異となって表れている．また，全てをセンターに集中するアプローチでは，安定を仮定していた通信線の事故による機能の停止となって表れている．

他方で，マルチエージェントによる分散処理の方法が提案されている．しかし，この系統の方法の恩恵に与るためには，決められたエージェントの仕組みに従ったプログラミングが必要である．従来型のプログラミングに慣れた大多数のプログラマにとって，これは大きな制約となる危険性がある．

こうした背景のもとに，IoT のプラットフォームとして，ネットワーク上に仮想的な計算機を構成する方法を提案する．もし，そういうことが可能であれば，プログラマは，ネットワーク上のプログラミングと単一計算機上のプログラミングとを区別する必要はない．また，センターと端末との非対称性もなくなることになる．

ネットワーク上の仮想的な計算機を構築するためには，ハードウェア，ソフトウェアの両方からサポートが必要である．ソフトウェアだけで仮想的な計算機を作ることは可能であるが，安全性と実用的な実行速度の確保のため，ハードウェアのサポートも重要な要素となる．

仮想的な計算機を構成する最低限の要素は記憶域と実行主体とである．より具体的には仮想化されたメモリとプロセッサ（スレッド）ということになる．ここでは，メモリとスレッドと呼ぶことにする．

本報告の以下のセクションでは，ネットワーク上でメモリとスレッドとを仮想化する方向に関して述べる．

2. 計算機の抽象概念

2.1 抽象の概要

ネットワーク上に仮想計算機を作るに際して，計算機という概念を抽象的に定義する必要がある．計算機の理論モデルとして知られているのはチューリングマシン（Turing Machine）[1] である．チューリングマシンは， 図 1 に示

[1] フリーランス・コンサルタント
Freelance Consultant, Japan
[a)] ken1maeda@ams.kuramae.ne.jp

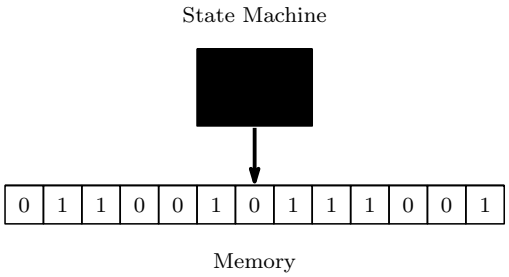

図 1 チューリングマシン
Fig. 1 Turing Machine

すように，メモリ（Memory）と状態機械（State Machine）とからなる．ただし，外界との入出力（I/O）は省略している．

ここには，二次記憶装置もネットワークもない．したがって，一番単純な計算機の概念として，この二つがあれば良いことになる．以降，状態機械をプロセッサと呼ぶことにする．

2.2 メモリの抽象概念

通常の計算機には，「主記憶（Main Memory）」と「二次記憶（Storage）」とが存在する．チューリングマシンでは，この区別がなかったことから，主記憶と二次記憶との二重構造は，計算機として本質的なものではないことがわかる．では，何故ほとんどの計算機では両方が存在するのであろうか．一つの理由は，速度と経済性との両立が求められるためである．主記憶は高速であるが高価で，二次記憶は低速であるが安価である．もう一つの理由は，電源を切った際の記憶保持の必要性のためである．したがって，これらの問題点が解決できるならば，記憶の二重構造が存在する意味はない．ただし，先に述べたように，入出力は別の問題 [*1] である．

主記憶と二次記憶との関係に関して，既に仮想記憶という概念があり，物理的な主記憶の容量以上に主記憶があるように見せかける技術がある．仮想記憶では，主記憶を実際より多く見せるために，二次記憶との入れ替えを行う．仮想記憶でない場合には，主記憶と二次記憶との入れ替えを明示的にプログラミングしなくてはならないが，仮想記憶では OS が裏で入れ替えを実行する．

オーソドックスな仮想記憶では，二次記憶も明示的に存在する．それは，先の述べた理由のうち，主として記憶の保持のためである．その二次記憶のデータにアクセスするためには，`READ/WRITE` のシステムコールを明示的に使う必要がある．しかし，記憶の保持を目的とした主記憶と二次記憶との二重構造は本質的なものではなかった．そこで，これらを区別しないようにする．そういう仮想記憶は，単一レベル記憶（Single-Level Store: SLS）[2], [3] と呼ばれている．SLS が最初に採用されたのは Atlas [4] である．しかし，より有名になったのは Multics [5] である．Multics は，全てのマシンがシャットダウンされており，現存する SLS システムは，AS/400 [6] の思想を受け継いだ IBM i である．

ここでは，メモリの抽象概念として単一レベルの記憶を考えることとする [*2]．

2.3 プロセッサの抽象概念

プロセッサは，チューリングマシンにおける状態機械であった．チューリングマシンでは，状態機械は一つしか描かれていないが，時分割（Time Sharing）[7] によって複数の仕事を並行して実行しているように見せかける技術が存在している．

これら複数の仕事は，タスク（Task），あるいはプロセス（Process）と呼ばれる．しかし，一つのプロセスに複数のスレッド（Thread）が存在する場合もあり，この場合はスレッドが最小の実行単位と見做す方が適切である．

Unix 系の OS では，プロセスとスレッドは明確に異なっている．しかし，その主因は，アドレス空間の切替えという Unix 系 OS の保護方式である．保護とスレッドの詳細とに関しては，別の節で議論する．

ここでは，プロセッサの抽象概念としてスレッドを採用する．

3. 仮想化の詳細

3.1 仮想化の概要

前節での議論から，計算機の仮想化にはメモリとスレッドを仮想化すればよいことがわかる．問題は，これらをネットワーク透過的に仮想化する方法である．ここで提案するのは，ネットワーク共有メモリと，それをベースとしたスレッド（およびスケジューリング）である．

3.2 メモリの仮想化

まず，単一レベル記憶について述べる．通常，主記憶には，アドレスと呼ばれる一次元の位置情報があるが，二次記憶には，それ以外にファイルという概念が導入されている．Multics では，それらを統合し，単一レベル記憶にセグメンテーション（Segmentation）[8] が使われた．すなわち，メモリのアドレスとして，<segment, offset> という二次元の指定ができた．それは，あたかも＜ファイル名，先頭からのオフセット＞というファイルのアクセスに似た概念である．

通常の一次元のアドレスと，セグメンテーションの差を図に示す．一次元アドレス（図 2）では，ある領域（たと

[*1] たとえば，マウス，カメラ，などの実時間入力を考えれば，メモリではない入出力は必須であることがわかる．

[*2] こういう考え方は，ストレージに半導体が利用されることが一般的になった今日において，より説得力が増しているように思われる．

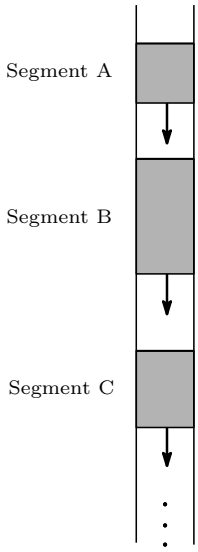

図 2 一次元アドレス
Fig. 2 Linear Address

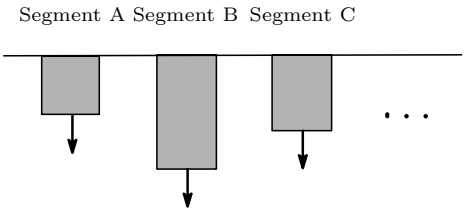

図 3 セグメンテーション
Fig. 3 Segmentation

えば Segment A）が大きくなると，別の領域（たとえば Segment B）に重なる危険性があるが，セグメンテーション（図 3）ではその恐れはない．

なお，図中は連続した領域として表示しているが，物理的にはその必要はない．ファイルがブロックに分割されていたり，仮想空間がページごとに分割されていたりするのは当然である．

また，ファイルには，ファイル単位の保護の概念も付属している．Multics では，セグメンテーションによって，その概念を主記憶にも拡張した．ここでも，それに倣うことにする．それによって，セグメント単位ごとにメモリの保護方針を変更することが可能となる．

プログラマから見ると，ファイル（実際にはメモリのセグメントであるが，ここではなじみ深いファイルと呼ぶ）に直接（`load/store` するように）アクセスしているように見える（全てのファイルをあらかじめ `mmap` したのと同様と考えるとわかりやすい．）．主記憶は，ファイルの（ページごとの）キャッシュとして働く．CPU のキャッシュが通常プログラマから見えないのと同様に，主記憶もプログラマからは見えない（あるいはファイルと同一のセグメントとして見える）．

ここでメモリの仮想化として実現したいのは，ネットワーク全体で共通のセグメンテーションのアドレッシングを持つものである．ネットワークの向こう側にあるファイルをアクセスする技術は，既に確立されている（たとえば，NFS [13]）．メモリとして見えるか，ストレージとして見えるかの差は，ここでは議論しないが，少なくともプログラマから見てファイル（セグメント）に見えるようであれば問題はない．理想的には，同じ対象物は，後述する共有のため，同じセグメントアドレスを持つのが好ましいのはいうまでもない．

この仮想化されたメモリは，ネットワーク上に仮想的に構築されるプロセッサによって共有される．すなわち，ネットワーク上の共有メモリ（Shared Memory）[9] ということになる．

過去の類似した共有メモリの例として，Dash [10] をあげることができる．しかし，Dash は，プロセッサの結合のために階層化されたバス構造を前提としており，ネットワークによる結合とは大幅に異なっている．バス構造の場合には，キャッシュコヒーレンシー（Cache Coherency）**?** の確保が比較的容易であるのに対して，ネットワークではほぼ不可能である．そこで，コヒーレンシーが必要な場合には，別のソフトウエア的な手当てを取ることとし，ハードウェアによるコヒーレンシーの確保を前提としないこととする．

メモリの共有と保護に関しては，後述する．

3.3 プロセッサの仮想化

スレッドがプロセッサの仮想化であることは，一般的な概念として論を待たない．ここでは，スレッドが所有すべきリソースの問題を考察する．

物理的なプロセッサには，以下のリソースが属している．

- 状態を表現する複数のフラグ（Flag）
- 汎用レジスタ（General Register (GR)）
- プログラムカウンタ（Program Counter (PC)）
- スタックポインタ（Stack Pointer (SP)）
- グローバル変数ポインタ（Global Variable Pointer (GVP)）
- スタティック変数ポインタ（Static Variable Pointer (SVP)）

グローバル変数ポインタとスタティック変数ポインタは，従来は兼用されており，また，明示的なポインタというより，汎用レジスタを用いることが多かった．ここで敢えてポインタとして出したのは，機能としてそういうものが必要であることを明確化するためである．

スレッドは，過去の OS でも，さまざまな概念でとらえられてきた．ここでは，通常の UNIX，その発展系である Mach，OS の名称にスレッドを関した Thread on Module（ToM）[12] と提案 OS のスレッドを比較してみる．

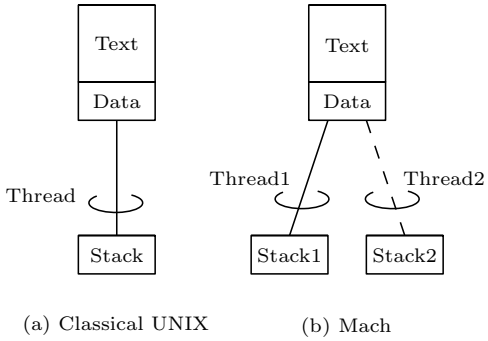

図 4 UNIX と Mach のスレッド
Fig. 4 Threads in UNIX and Mach

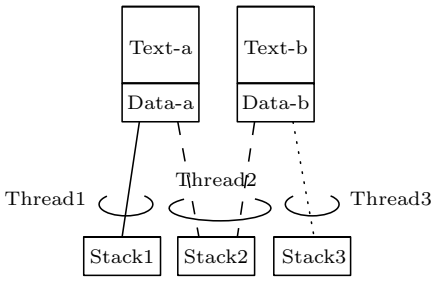

図 5 ToM のスレッド
Fig. 5 Threads in ToM

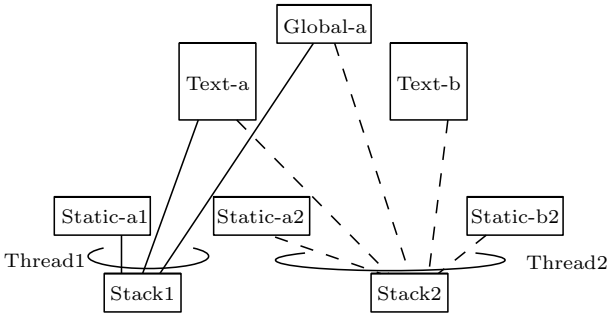

図 6 提案 OS のスレッド
Fig. 6 Threads in the Proposed OS

古典的な UNIX スレッドは，プロセスと密接に結びついていた（図 4(a)）．Mach では，UNIX プロセスに相当する中に複数のスレッドがあることができるようになった（図 4(b)）．ToM では，スレッドは，複数のモジュール（図中は Text）を飛び歩けるようになった（図 5）．提案 OS におけるスレッドは，ToM に最も近いが，さらにグローバル共有データ（図中は Global）とスレッドごとのスタティック変数（図中は Static）を持つことができるようになっている（図 6）．

次の問題は，それをいかにしてネットワーク透過にするかということである．ここでの基本的なアイディアは，前節で考察したネットワーク上の共有メモリを利用することである．より具体的には，スレッドを管理するテーブル

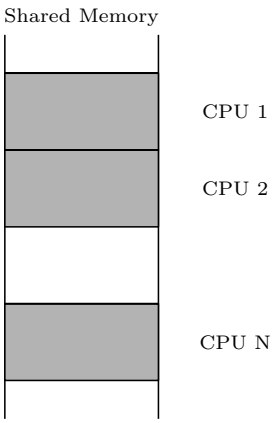

図 7 スレッドテーブル
Fig. 7 Thread Table

をシンプルに共有メモリ上に置くことである．スレッドのテーブルとしては共有されるが，物理的な CPU に実行を割り当てる際に，特定の CPU が特定のアドレス領域から選択するようにする．その様子を図 7 に示す．

スレッドを特定の CPU で動かす場合には，その CPU に対応するテーブルにスレッド情報を移動することにより，CPU の割り当てが可能となる．実行すべき CPU を選択するのは，スケジューラ（Scheduler）である．スケジューラは，単に CPU の空き状況だけでなく，スレッドを実行する際のコストを計算することにより，割り当てるべき CPU を決定する．

もちろん，自分の CPU 上で勝手なスレッドに動かれては困る場合もある．それを制御するためには，自分の CPU が管理するスレッドテーブルに適切な保護を設定することである．スケジューラの詳細および保護の問題は，後述する．

4. メモリの保護と共有

4.1 メモリ保護

セグメンテーションの採用により，バッファーの境界を越えた別の領域の破壊に対して大幅に強くなっている．ただし，それだけでは十分な保護とは言えない．

ファイルごとに保護の設定があったように，単一レベル記憶でもセグメントごとに保護の設定が必要である．UNIX あるいは類似システムでは，ファイルに対してアクセス制御リスト（ACL）をセットすることができる．それは，`user (u)/group (g)/ other (o)` に対して，`read (r)/write (w)/execute (x)` で表される．（ディレクトリに関する ACL は，スレッド関係で一部議論する．）保護の相手側の設定が，`u/g/o` で良いのかという問題もある．たとえば，Windows では，相手側をもっと細かく設定できるようになっている．Multics でも，相手側を細かく設定することができた．しかし，ここでは，この問題への深入

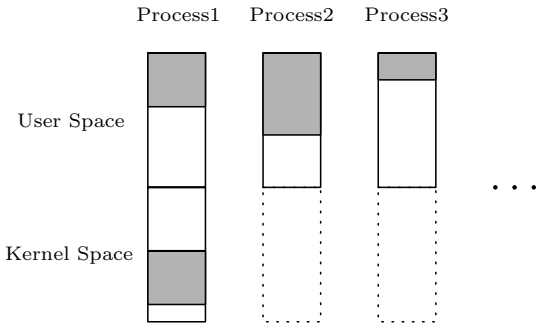

図 8 UNIX の仮想空間
Fig. 8 Virtual Space of UNIX

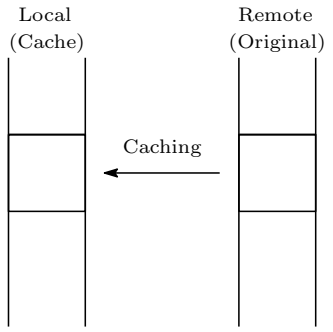

図 9 キャッシュ
Fig. 9 Caching

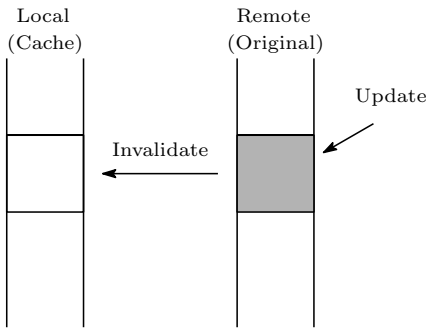

図 10 無効化
Fig. 10 Invalidate

りはしないこととし，セグメントごとに ACL があるということを指摘するにとどめる．

UNIX のアクセス制御は，ファイルに対してであって，メモリに対してではない．メモリに対しては，ユーザ領域とシステム領域を分けることと，プロセスのユーザ領域空間を切り替えることによって保護を実現している（図 8）．

しかし，これは別のプロセスがアクセスしないようにしているだけで，自分自身のプロセスは，どこでも自由にアクセスできてしまう．（テキスト領域は読み込み専用にする，などの保護は可能.）提案方法では，メモリをセグメント化し，セグメントごとに ACL を設定しているため，自分自身であっても書き込みができないメモリ領域を作ることができる．

なお，Multics ではリング保護（Ring Protection）も採用されており，さらに強力なメモリ保護が実現されていた．リング保護は，本稿の範囲を越えるので，別の機会に議論することとする．

4.2 メモリ共有

4.2.1 メモリ共有の概要

前提条件として，必要な情報ソースは，ネットワークの「向こう側（リモート（Rmote））」にあったとする．「こちら側（ローカル（Local））」のメモリ（主記憶・ストレージ）は，キャッシュ（Cache）という位置付けになる（図 9）．したがって，ここで述べる問題の本質は，キャッシュコヒーレンシーの問題と同じである [*3]．

高速のバスやスイッチで共有された状況とは異なり，ネットワーク経由で共有された場合には，コヒーレンシーに問題が生じる危険性が高い．ここでは，共有に関する注意事項と，ソフトウェア的な対処に関して述べる．

4.2.2 読出し専用のケース

一般的に，データアクセスのほとんどは，データの読出し（`READ`）であり，書込み（`WRITE`）は少ない．書込みがないデータを読出し専用（`Read Only (RO)`）と呼ぶ．読出し専用データに関しては，そのデータが最新のものであるかどうかのみに注意すればよい．そのため，キャッシュコヒーレンシーの問題は，比較的軽い．読出し専用データの管理ポリシーとして，リモートのデータを利用する際にローカルにキャッシュし，ACL は `r` のみをセットする．

リモートの大元データが更新された際には，大元を管理するノードから，キャッシュを無効化（Invalidate）するように指示する．この様子を図 10 に示す．

4.2.3 書込みのケースとバリエーション

読出し専用の場合には，前節に述べたように比較的簡単な処理で間に合った．しかし，書込みがある場合には，問題は簡単ではない．まず最初に，書き込むケースのバリエーションを考えてみる．

- 書き込むのは自分ひとりの場合
- 自分専用の修正を作る場合
- 複数の人が修正する場合（トランザクション（Transaction））

書き込むのが自分ひとりの場合は，自分がデータの管理者であり，修正は管理者に任されているケースである．この場合には，自分が書き込んだデータを大元の場所（*home*）に書き戻し（Write-back），自分以外のキャッシュを無効化する．

[*3] 全ての場合ではないが，マルチプロセッサにおける Cache-Only Memory Architecture（COMA）[14]，NUMA[15]-COMA を参照．

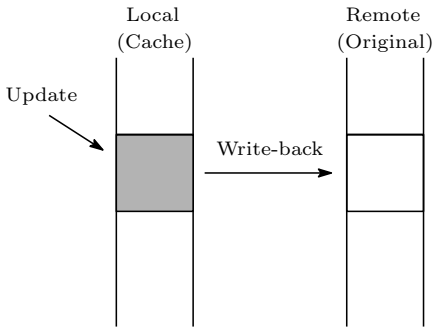

図 11 書き戻し
Fig. 11 Write-back

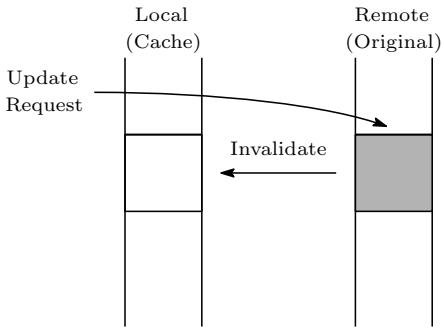

図 13 トランザクション
Fig. 13 Transaction

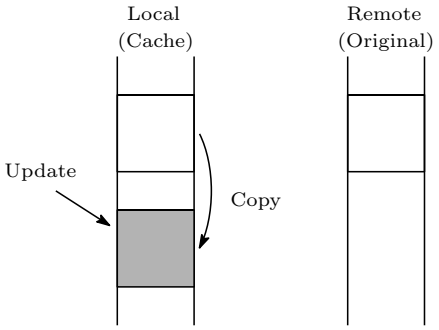

図 12 ローカルバージョン
Fig. 12 Local-versions

書き戻しの様子を図 11 に示す．なお，無効化は，既に図 10 に示している．当然，自分に関する ACL は，`r/w` がセットされており，他の人は，`r` のみがセットされている．

自分専用の修正を作る場合は，複数のローカルバージョン（Local-versions）を許すケースである．図 12 に示すように，書き込もうとした際に元のデータをコピーし，コピー後のメモリに書き込む．元のデータの ACL は，`r` のみがセットされており，コピー後のものは，`r/w` がセットされる．名前空間（Name Space）の管理方法にも依るが，全体で共有しているとすると，新しいバージョンは，別の名前（たとえば，元の名前が *foo* であった場合，*foo*(1)）にするなどの対策が必要である．

複数の人が修正する場合は，トランザクション（Transaction）のケースである．これは，ローカルのキャッシュを勝手に修正すると困ったことになるので，大元を管理するサーバー（Server）に依頼する方法を採らざるを得ない．ローカルには，必要であればキャッシュを置くことができるが，修正のたびに無効化されることになる．この動作を図 13 に示す．なお，無効化は，全てのキャッシュに対して行われる．

5. スケジューラ

通常，スケジューラは，単一の計算機上でリソースや実行時間や，場合によっては優先順位，などを考慮して実行するスレッドを決定する．ここでは，前節までに述べたように，ネットワーク上に仮想的な計算機を構成したため，スケジューラもネットワークを跨ぐように構成する必要がある．

仮想計算機としては，ローカルとリモートとの論理的な差はないのであるが，物理的にはいろいろな差が存在する．たとえば，

- 物理的な計算能力
- （キャッシュされた）データの有無
- 通信にかかる時間
- あるプロセッサが物理的に扱うスレッドの数

などである．

これらを，「コスト（Cost）」と見做して，コストが最小になるように実行するスレッドを決めるというのが，基本的なアイディアである．以下に，いくつか具体的な例をあげてみる．

5.1 スタンドアローンのケース

通信は頑健で十分に高速であると仮定されることがあるが，実際には，回線故障や混雑によって切れたり遅くなったりする．通信回線が切れていたり遅い場合には，全てをローカルで実行せざるを得ない．

こういうケースでは，通信のコストとして大きな値になるため，コスト計算によって，自動的にスタンドアローンで動くことになる．データがリモートにしかない場合には，どのような方法でも動くことができないので，ローカルに読出し専用でキャッシュされていると仮定している．

5.2 クラウド指向のケース

一般的に，モバイル端末よりクラウドの方が計算能力が高い．ネットワークの遅延が気にならないケース（転送すべきデータ量が少なく，リアルタイム性が低い場合）では，クラウド側に全ての処理を任せる方が良い．

この場合，主要なコストは，［通信速度 x データ量 + ス

レッドあたりのプロセッサ速度 x 計算量］の値で，クラウド側が勝る（値として小さくなる）ため，自動的にクラウドコンピューティングとなる．使うデータ量が多く，特定のノードにそのデータが（キャッシュされて）存在する場合には，そのノードで処理も実行することとなる．

5.3 自動車応用のケース

「自動運転」あるいは「自立運転」が注目されているが，こういう場合には，ローカルとリモートとの両方での処理分担が必要になると想定される．

障害物を検知してブレーキをかけたりハンドルを操作したりするのは，リアルタイム性が必須である．また，カメラなどの車載センサーから得られる時々刻々のデータも，ローカルにある．したがって，この処理はローカルで実行せざるを得ない．

一方，道路の混雑状況とか，交差点の死角から進行してくる車の情報とかは，ローカルにはなく，どこかのサーバーが管理しているはずである．したがって，ナビゲーションや複数の車が関係する警告の処理は，クラウド側で実行する方が良い．

5.4 ダイナミックな変更のケース

ここまでの例は，処理を固定的に割り付ける場合ばかりであったが，通信の状況も，あるノードで実行されているスレッドの数も変化している．スケジューラは，定期的にコスト計算を行い，ダイナミックに割り当てを変更することが好ましい．

たとえば，ローカルでスタンドアローンで動いていた状況で，急に通信が可能になった場合には，続きはクラウド側に任せる方が良いかも知れない．その逆は，実際には難しい場合が多い．突然通信が途絶えた場合には，クラウド側からローカルに実行を移すこともできなくなる．しかし，ローカルに残った情報を使って，再計算したり，とりあえずの処理のみローカルで実行することは不可能ではない．

5.5 ノードでの実行制御

バッテリーが乏しくなったモバイル端末や，いろいろな理由から，あるノードで他所からのスレッドを動かしたくない場合がある．ここで，スレッドを別の CPU で稼働させる仕組みを思い出すと，図 7 に示したように，あるメモリの領域が CPU に対応しており，CPU n でスレッドを実行させるためには，スケジューラが，CPU n に対応する領域にスレッド情報を書き込めば良かった．逆に，その領域の ACL で w をセットしなければ，スケジューラはその領域に書き込むことはできない．

スケジューラが書き込んで良いかどうかは，ユーザが権限を持っているかどうかによって決定される．スケジューラの詳細は，ここでは省略するが，仮想計算機にログイン（Login）[*4] しているユーザの権限と，前述の ACL とを反映して特定のノードでの実行を決定するメカニズムが必要である．

6. ネットワーク上の仮想計算機が作る世界

従来は，ローカルに限定したコーディングと，クラウド指向のコーディングとを区別して考えており，通常は別のコーディングが必要であった [*5]．本稿では，従来と異なるアプローチとしてネットワーク上に仮想計算機を構成して，その計算機のコードを書くという方法について提案した．ネットワークを意識するのは，プログラマではなく，スケジューラである．

ユーザは，自分のリソース（データ，プログラム，など）に対して，どういう形式で共有を許すかをセットすることによって，実行形式を制御することになる．したがって，本稿で提案する方法では，ネットワーク特有の問題をコーディングからリソースの属性に移し替えるものであると言うことができる．

その中で，トランザクションは特殊なケースであり，現状では，やはりトランザクション用のコードを書かなくてはならない．それを区別しない方法も可能であると考えられるが，本稿の範囲を逸脱するため，別の機会に議論する [*6]．

7. おわりに

本稿では，IoT 時代のソフトウェア開発のプラットフォームとして，ネットワーク上に仮想的な計算機を構築する方法について述べた．このようにして構成された仮想計算機で，普通に単独計算機のマルチコアやマルチスレッドを使ったコーディングをすることにより，ネットワーク上のアプリケーションを簡単に作成することが可能となる．

実は，各プロセッサで同じコードが動く必要があるのであるが，仮想コードが使える仮想計算機にすることにより，その問題も解決可能である．また，コードモーフィングという技術も知られており，実際に，たとえば，Crusoe では，コードモーフィングによって X86 のコードを実行することが可能であった [16]．

全世界のネットワークで，同じ仮想アドレスが使えるようにしようとすると，アドレス空間の小ささも問題となる．セグメンテーションをハードウェアでサポートする Intel X86 の 32 bit モードでは，16 TB の仮想空間しかアクセスすることができない．

[*4] ネットワーク透過な仮想計算機へのログインメカニズムも重要な問題であるが，ここでは別の機会に譲る．

[*5] ローカルにあっても無理に通信を使えば，原理的には同じコーディングが可能である．

[*6] オブジェクト指向でのコーディングを前提とするならば，それほど困難な問題ではない．しかし，冒頭で述べたように，従来のコーディングの枠組みとは異なるものになる．

これらをどのように解決していくかということは，今後に残された問題である．

参考文献

[1] Turing, A.M.: On Computable Numbers, with an Application to the Entscheidungs Problem, *Proc. of the London Mathematical Society*. Ser. 2, Vol. 42 (1937), pp. 230–265.
[2] Wikipedia: Single-level store, available from ⟨https://en.wikipedia.org/wiki/Single-level_store⟩, (2016-09-05).
[3] Shapiro, J.S. and Adams, J.: Design Evolution of the EROS Single-Level Store, *Proc. of 2002 USENIX Annual Technical Conference*, (2002), pp. 59–72.
[4] Kilburn, T., et al: One-Level Storage System, *IRE Trans. Electronic Computers*, Vol. 2 (1962), pp. 223–235.
[5] Organick, E.I.: *The Multics System*, MIT Press (1972).
[6] Soltis, F.G., *Inside the AS/400*, 29th Street Press (1997).
[7] Wikipedia: Time-sharing, available from ⟨https://en.wikipedia.org/wiki/Time-sharing⟩, (2016-09-05).
[8] Wikipedia: Memory segmentation, available from ⟨https://en.wikipedia.org/wiki/Memory_segmentation⟩, (2016-09-05).
[9] Nitzberg, B. and Lo, V: Distributed Shared Memory: A Survey of Issues and Algorithms, *IEEE Computer*, Vol. 24 (1991), No. 8, pp. 52–60.
[10] Lenosky, D., et al: The Stanford Dash Multiprocessor, *Computer*, Vol. 25, No. 3, (1992), pp. 63–79.
[11] Wikipedia: Cache coherence, available from ⟨https://en.wikipedia.org/wiki/Cache_coherence⟩, (2016-09-05).
[12] Hagino, T. and Yamagishi, K.: The ToM (Thread on Modules) Microkernel, *Systems and Computers in Japan*, Vol. 24 (1993), No. 9, pp. 14–21.
[13] Wikipedia: Network File System, available from ⟨https://en.wikipedia.org/wiki/Network_File_System⟩, (2016-09-05).
[14] Wikipedia: Cache-only memory architecture, available from ⟨https://en.wikipedia.org/wiki/Cache-only_memory_architecture⟩, (2016-09-05).
[15] Wikipedia: Non-uniform memory access, available from ⟨https://en.wikipedia.org/wiki/Non-uniform_memory_access⟩, (2016-10-10).
[16] Wikipedia: Transmeta Cruesoe, available from ⟨https://en.wikipedia.org/wiki/Transmeta_Crusoe⟩, (2016-10-04).