卒業論文

グリッド計算システム AgentTeamwork における計算機資源の監視機能の開発に関する研究

愛媛大学工学部情報工学科

平成 14 年度入学

森崎 潤

指導教員

小林 真也 教授

平成 18 年 2 月 15 日提出

概要

グリッド計算とは、インターネット上に点在する多数のコンピュータを結ぶことで仮想的に高性能な計算機を作り出し、ユーザがそこから必要な処理能力や記憶容量を利用することを可能にするシステムである。複数のコンピュータに並列処理を行わせることで、高速に大量の処理を実行することが可能である。

AgentTeamwork システムは、グリッド計算システムの一つであり、ワシントン大学バセル校にて開発されている。AgentTeamwork システムでは、ネットワーク上の計算機間を自律的に移動して、資源を活かしたサービスを実現するモバイルエージェントを用いて、ユーザジョブをグリッド環境上に点在する負荷の軽い計算機へ運び、処理させることができる。負荷の軽い計算機を把握するため、AgentTeamwork システムは、モバイルエージェントを利用して各計算機資源の利用状況を監視する。しかしながら、現在の AgentTeamwork システムは、資源情報を更新するエージェントにかかる負荷の過多、更新頻度の低下、ジョブの並列処理の性能向上に直接利用できる資源をすべて測定しているとは言えないという問題点を含んでいる。

そこで、本研究では、これらの問題点の解決のため、新たに資源を監視するモバイルエージェントを開発し、そのモバイルエージェントが再帰的にエージェントを生成して、対応するエージェント間において、ネットワーク帯域幅を含む種々の資源を調査するアルゴリズムを実装した。また、そのエージェントに移動の機能を付加することで、所定の計算機だけではなく、複数の計算機を移動しながら調査するアルゴリズムも実装した。これらのアルゴリズムの実装により、上述した問題点の解決に成功した。本論文では、そのアルゴリズムを解説し、AgentTeamwork システムの資源監視機能における信頼性および性能を評価する。最後に、本研究での成果および今後の改善点について述べる。

目 次

第1章　緒論

過去10年間、モバイルエージェント [1][2] は、情報収集、電子商取引、ネットワーク管理のために適用される分散システムの将来的な基盤として注目を浴びてきた。これは、複数のモバイルエージェントを分散システムに適用することで以下の利点を得ることができるからである。

- グリッド環境上に点在する計算機資源を探索できる
- ジョブを、ある計算機から別の計算機へ移動させることができる
- 異なる計算機上で走るクライアントプロセスとサーバプロセス間で発生する通信のオーバーヘッドを減らすことができる
- モバイルエージェントに、負荷の軽い計算機を利用する機能を実装し、効果的にジョブを処理させることができる

上記の利点にもかかわらず、モバイルエージェントを適用した分散システムは、他の営利なグリッド計算システムと比べて、優位性が見当たらない。これは、そのシステムが、マスターワーカーモデルから完全に逸脱できていないからである。このため、モバイルエージェントは、計算機へジョブを配置する、および、ジョブの処理結果を回収する目的で、単調に利用されている。

これに対して、本研究で実装中の AgentTeamwork システムは、ジョブの投入や処理のために中心となるサーバを必要とすることなく、モバイルエージェントをユーザジョブの並列分散処理に必要な各種の作業（MPI ランクの割当て、プロセス間通信の確立、異なるクラスタをまたがるプロセスの配置、およびジョブの移動）、および、高いフォールトトレランスを実現するために利用する [2]。AgentTeamwork システムを利用するユーザは、彼らの計算機をグリッド環境上の計算機資源として登録するため、資源情報を共有 ftp サーバに保存しなければならないが、これ以外の処理は、すべて分散して行われる。モバイル

エージェントは、その ftp サーバから計算機資源の情報を取得すると、以下の処理を並列に行うことができる。

(1) 遠隔の計算機の資源状況を定期的に調査する

(2) ジョブ処理のために、最適な資源を持つ計算機を選択し、そのジョブを割り当てる

(3) ジョブの実行状況を監視および把握する

(4) 負荷の軽い計算機へジョブを移動させる

(5) 最新のスナップショットを用いて、ジョブを再開する

上記の (1) に関して、現在の AgentTeamwork システムでは、次のように資源の調査を行う。資源情報を更新するエージェントが、AgentTeamwork システム内で設定されている数のエージェントを生成し、そのエージェントを調査対象の計算機へ割り当てる。計算機へ割り当てられた各エージェントは 5 分間隔で資源を調査し、資源情報を更新するエージェントにその結果を送る。しかし、この調査方法には、以下の問題点が含まれている。

- 計算機資源の情報を更新するエージェントにかかる負荷が過多である
- 情報の更新頻度が低くなる場合がある
- ジョブの並列処理の性能向上のために直接利用できる資源を監視していない

本研究では、これらの問題の解決を目的する。はじめに、資源情報の更新を行うエージェントが、資源調査を目的としたエージェントを 2 つ生成するように改変した。この生成されたエージェントが再帰的に同じエージェントを生成し、対応するエージェント間で、ネットワーク帯域幅を含む種々の計算機資源を調査するアルゴリズムを実装した。さらに、このエージェントに移動の機能を付加することで、所定の計算機のみならず、複数の計算機を移動しながら調査するアルゴリズムを実装した。これらのアルゴリズムの実装により、上記の 3 つの問題点を解決することに成功した。本卒業論文では、上述のアルゴリズムについて解説する。また、アルゴリズム実装後の AgentTeamwork システム、および、既存の資源監視システムを用いて、動的に変化する計算機資源を調査する実験を行った。本論文は、両者の結果を比較して、本アルゴリズムの信頼性、および、性能について評価する。

本卒業論文は、以下の構成で示されている。第 2 章で、資源監視が可能な 3 つの既存のシステムを例示し、これらのシステムが現存しているにもかかわらず、AgentTeamwork システムに適用しない理由を解説する。第 3 章では、AgentTeamwork システムの概要および構成を述べ、現在、そのシステムが抱える資源監視の問題点を記載する。第 4 章で、本研究で実装した資源監視アルゴリズムを解説し、資源監視方法の改善点について考察する。第 5 章では、実装したアルゴリズムと既存の資源監視システムを比較する実験を行い、その結果を考察する。最後に、第 6 章で、本研究における結論を述べる。

第2章　関連研究

本卒業研究に関連するプロジェクトとして、資源管理の観点から Network Weather Service、Legion、Condor の 3 つのグリッド計算システムの特徴について、議論する。最終節では、本研究で実装中の AgentTeamwork システムが目標とする資源監視機能とそれらのシステムを比較する。

2.1　Network Weather Service

Network Weather Service（NWS）[5][6][7] は、ネットワーク、および、計算資源の情報をユーザが指定した間隔で周期的に監視し、それらの計測値に基づいて将来の値を予測する分散システムである。NWS には資源監視のためにネームサーバ、メモリサーバ、センサーという 3 種類のプロセスを使用する。ネームサーバは、センサーおよびメモリサーバのプロセスが起動しているホストの情報を一元管理するプロセスである。メモリサーバは計算機資源の情報管理を行うプロセスである。センサーは、実際に CPU や帯域幅等の各種情報を定期的に計測するプロセスである。測定の間隔は、ユーザが測定を開始させるコマンドを実行する際に、引数として秒単位で指定しなければならない。帯域幅は、ユーザにより指定されたセンサー間でデータを送受信し、送受信に要した時間と送受信したデータのサイズから算出される。センサーは測定した値を、直接メモリサーバへ送信し、メモリサーバはそれを保存する。ユーザはネームサーバやメモリサーバに問い合わせることでセンサー情報や各種計測値が得られるようになっている。

2.2　Legion

Legion は、デスク上のパソコン・ワークステーションから、このシステムがインストールされた計算機の資源にアクセスできるオブジェクトベースのメタシステムである [8][10][11]。

Legion システムは 5 つのオブジェクトにより成り立っている。そのうち、Collection、および、host というオブジェクトが計算機資源の管理に関わっている [9][12]。前者は計算機の資源情報の管理を行い、ユーザが指定した一意の計算機上に一つ存在する。そして後者は Legion をインストールした全ての計算機に存在し、資源の監視を行う。各計算機における host から、計算機資源の情報を Collection に回収させるためには、ユーザがコマンドを用いて、その host を資源回収の対象計算機として登録する必要がある。登録後は、定期的に資源の情報が Collection によって回収される。この情報の回収の間隔は、ユーザ自身の指定により変更可能である。デフォルトでは 300 秒であり、ユーザはコマンドを用いて秒単位で指定することが可能である [13]。ユーザは、シェルを使って一連のコマンドを実行することが可能だが、その場合、動的なホスト登録ができなくなる。また、Legion はチェックポイント作成のためのライブラリをユーザに提供しており、ユーザ自身がチェックポイントを作成しなければならない。

2.3 Condor

Condor は計算機の空き時間を、計算量の多いジョブの実行に利用することを目的としたジョブスケジューリングシステムである [14][16]。Condor において計算資源として割り当てられた計算機（Condor をインストールしている計算機であり、これらの計算機を Condor pool と呼ぶ）の内、軽負荷のものを計算資源としてジョブの処理に適用させる。Condor では、計算機の負荷の状態を知るために、計算機の利用可能な資源情報を確認する機能が必要である。そのため、Condor システムには、ユーザによって一意に指定された Central Manager と呼ばれる計算機が存在し、この計算機上で Collector と呼ばれるデーモンが、資源の一元管理を行う。また、Condor がインストールされた各計算機には startd と呼ばれるデーモンが存在し、このデーモンが各ホストの資源情報を ClassAd というファイルに記して、定期的に Collector へ送る。その頻度は、デフォルトで 300 秒であるが、これはユーザからも秒単位で指定可能である [15]。Collector によって回収されたファイルが、Central Manager 上で資源情報の管理に利用される。

また Condor では、ジョブの実行のチェックポイントを一元的に管理する Checkpoint server というデーモンが一つの計算機上に存在し、各計算機からチェックポイントがこの Checkpoint server へ送られる。この時、チェックポイントが受け取られるために要した遅延時

間が、Condor において監視対象資源の一つであるネットワークの性能を知る手段である[17][18]。

2.4 問題点

NWS は、それを実行する場合にユーザが度々コマンドを打ち込む必要がある。ネームサーバ、メモリサーバ、センサーの起動の際、さらに、監視したい資源一つ一つ（CPU やメモリ等）に対しての監視を開始するためのコマンドが必要となる。つまり、全てのプロセスはユーザのからのコマンド入力により立ち上げられる。ゆえに、もし NWS を自律的に起動させ自動的に資源監視を行わせるならば、各モバイルエージェントに異なるホスト上でコマンドを実行させる実装を施す必要がある。一方、AgentTeamwork システムは、ジョブを投入した後は、ユーザからの指示がなくともエージェントが自律的に処理を行うことを目標としている。

Legion においても、NWS 同様、コマンドを実行しシステムを立ち上げる必要がある。ゆえに、シェルで一連のコマンドを自動実行するか、さらに動的な監視を行う場合には、モバイルエージェントに行わせる必要がある。

Condor では、第 2.3 節で記したように、監視可能な資源の一つに、ネットワークの性能があり、ネットワークの遅延時間を求めることにより、それを知ることができる。しかし、その測定は Checkpoint server とチェックポイントを送信した計算機間のみである。Condor では、ジョブに従事するプロセス間で通信を行うにもかかわらず、そのプロセスが走っている計算期間のネットワークについての情報が得られない。

第3章　AgentTeamwork

本章では、はじめに第 3.1 節において、AgentTeamwork システムについての概要を示し、その働きについて解説する。第 3.2 節で、AgentTeamwork におけるモバイルエージェントの構成について述べる。第 3.3 節で、本システムが抱えている計算機資源の監視機能における問題点を挙げる。本章で用いられる図は、すべて本研究を行う前のシステム構成を表している。

3.1　概要

図 3.1.1 を用いて、AgentTeamwork システムへのユーザのジョブの投入から、計算結果が返ってくるまでの概略を示す。ユーザがジョブの処理を本システムに依頼したい時には、はじめにユーザ自身がコマンダエージェントを生成し、ジョブのプログラム名と引数をコマンダエージェントに渡す。コマンダエージェントはリソースエージェントを生成し（矢印 1)、共通の情報を格納した ftp サーバから、新たに登録された計算機の IP アドレスを取得し、さらに、ローカルのリソースデータベースを検索して、ジョブの処理に適した計算機の IP アドレスを選択する。リソースエージェントは、この情報をコマンダエージェントに渡す（矢印 2)。その後、リソースエージェントは、インターネット上に点在する各計算機の資源を監視し、定期的にローカルのリソースデータベースへ計算機資源の情報を更新する。コマンダエージェントは、センチネルエージェントとブックキーパエージェントを生成する（矢印 3)。センチネルエージェントは、ユーザジョブを選択された各計算機へ運び、そこで実際に処理をする。また、偶発的な障害でその処理が中断されてしまった場合や、より軽負荷の計算機にジョブを移動し、処理を行う場合を考慮して、処理の再開に必要な実行のスナップショットを採り、これをブックキーパエージェントに送る（矢印 4)。ブックキーパエージェントはそのスナップショットを保存する（矢印 5)。最終的に、ジョブの処理が完了すると、センチネルエージェントはその計算結果をコマンダエージェント

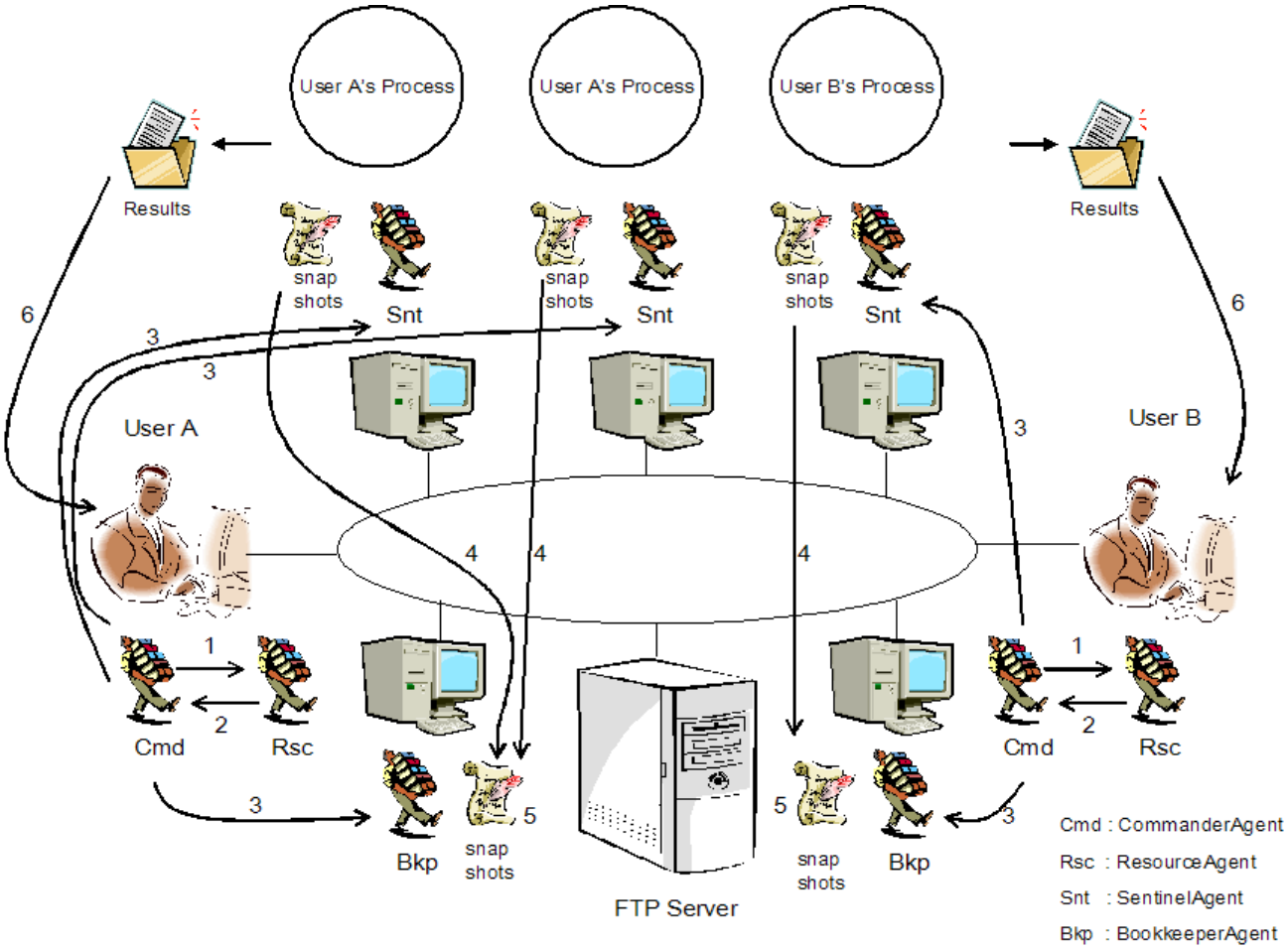

図 3.1.1: AgentTeamwork システムの概要

に返し、コマンダエージェントがユーザにその結果を返す（矢印 6）。

3.2 構成

図 3.2.1 に示すように、AgentTeamwork におけるモバイルエージェントの構成は、木構造を成す。各円がエージェントを表し、さらに、その円内の文字がエージェントの種類、Id に付随する数値がエージェントの識別子（以下 Id とする）を表す。コマンダエージェントがリソース、センチネル、ブックキーパエージェントを一つずつ生成し、さらに、コマンダエージェント以外の各エージェントが、階層的に子エージェントを生成する。子供のエージェントは、同じレイヤにおいて最も小さい Id を持つエージェントから順に、予め設定された数（現在、4 に設定されている）の子エージェントを生成する。センチネルエージェン

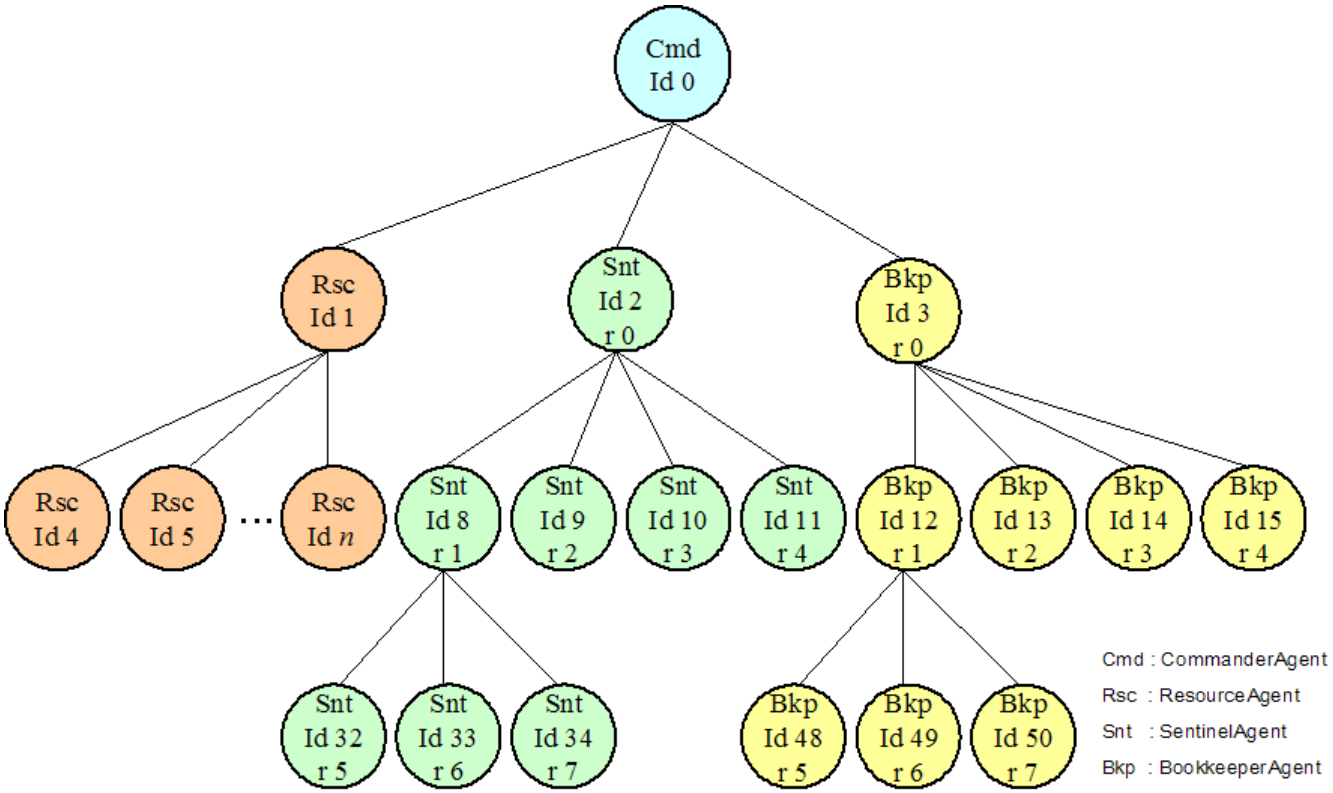

図 3.2.1: AgentTeamwork におけるエージェントの構成

トとブックキーパエージェントに記載されている r の数値は MPI ランクを表す。それぞれの MPI ランクは式 3.1、式 3.2 により求めることができる（式内の rank は MPI ランク、sId はセンチネルエージェントの Id、bId はブックキーパエージェントの Id、N は各エージェントが生成できる子供の数を表す）。センチネルエージェントとブックキーパエージェントは同じ数だけ生成される。これは、各センチネルエージェントが採った実行のスナップショットを保存するために、同じ数だけのブックキーパエージェントを必要とするためである。センチネルエージェントは、実行のスナップショットをブックキーパエージェントに送る際、式 3.3（式内の bId はブックキーパエージェントの Id、sId はセンチネルエージェントの Id、N は各エージェントが生成できる子供の数を表す）を用いて、スナップショットを保存するブックキーパエージェントの Id を算出することができる。

$$rank = sId - (2 + (2N - 3) \sum_{i=0}^{\frac{ln(sId/2)}{ln(N)} - 1} N^i) \tag{3.1}$$

$$rank = bId - (3 + (3N - 4) \sum_{i=0}^{\frac{ln(bId/3)}{ln(N)}-1} N^i) \tag{3.2}$$

$$bId = sId + N^{\frac{ln(sId)}{ln(N)}} \tag{3.3}$$

リソースエージェント [3][4] は、計算機資源の監視のため、子リソースエージェントを生成する。子供のリソースエージェントは、各ノード上で資源の情報を得るための調査を 5 分間隔で行い、親のリソースエージェントに調査結果を報告する。

センチネルエージェントは、ユーザから要求のあったジョブを処理するために必要な数だけ自分の子供を生成する。必要な数のセンチネルエージェントが生成されるまで、生成された子供が、さらに、4 つずつ子供を生成する。その後、各センチネルエージェントは自分の請け負ったジョブの処理を開始する。実行結果は木構造の葉にあたるセンチネルエージェントから根に位置するコマンダエージェントへと回収されていく。このアルゴリズムを適用することで、コマンダエージェントは必ず、すべてのセンチネルが処理を完了してから、ジョブの終了を知ることになる。

ブックキーパエージェントは、センチネルエージェントから送られてきたスナップショットを受け取ると、そのエージェントのランク値が $rank$ である場合、$rank - 1$ と $rank + 1$ の値を持つ別のブックキーパエージェントにも、そのスナップショットを転送する。これは、あるセンチネルエージェントのスナップショットを保持するブックキーパエージェントが偶発的な障害により、停止した場合でも、別のブックキーパエージェントから同一のスナップショットを回復することができ、より障害に強い設計の実現を可能にするためである。

3.3 計算機資源の監視機能の現状

計算機資源の監視機能の点から AgentTeamwork の問題点を観察すると、以下の 3 つの問題点に絞られる。

(1) N 台の計算機を監視するため、リソースエージェントが N 個の子リソースエージェントを生成した場合、親のリソースエージェントはすべての子供から、計算機資源の

監視結果を受け取らなければならず、1 対 N の対応を強いられるため、大きな負荷がかかる。

(2) 各エージェントが生成可能な子供の数が n であり、n よりも N の値が十分大きい値であるとき、各リソースエージェントは N/n 台の計算機を移動しながら監視する必要がある。この場合、一つのリソースエージェントが担当する計算機の台数が多すぎると、各計算機の資源情報の更新頻度が低くなる可能性がある。更新頻度を高くするためには、n の値も増大する必要があるが、逆に親のリソースエージェントの負荷も増大するので、(1) での問題点が解決されない。

(3) リソースエージェントは、親のエージェントがいるユーザ自身の計算機と子供のエージェントが移動した先の遠隔の計算機との間にあるネットワークの帯域幅を測定することができるが、遠隔の計算機間の帯域幅は測定しないので、並列アプリケーションの子プロセス間通信の性能向上に直接利用することができない。

資源監視の機能における、以上の問題を解決するため、監視能力の向上を図るアルゴリズムを実装した。次章において、その内容を説明する。

第4章　センサエージェントの実装

本章では、第 4.1 節で、資源を調査するエージェントが、割り当てられた計算機上で定常的に資源を監視するアルゴリズムについて解説する。第 4.2 節では、前節のアルゴリズムを改良し、エージェントに計算機間を移動させ、多数の計算機の資源を監視するアルゴリズムについて解説する。第 4.3 節で、AgentTeamwork システムが監視することのできる測定項目を示す。第 4.4 節で、実装したアルゴリズムによる資源監視機能のさらなる改良方法について考察する。

4.1　静的監視アルゴリズム

第 3.3 節で示した問題点を解決するため、リソースエージェントを改良した。図 4.1.1 に、その改良を表す（円内の数字は Id を表す）。リソースエージェントは、センチネルエージェント、および、ブックキーパエージェントと同様に、再帰的にセンサエージェントを生成し、そのセンサエージェント郡に計算機資源の監視を行わせる。帯域幅の測定は、木構造において対応関係にあるセンサエージェント間で行われ、一方がデータを送信、もう一方がデータを受信する役割を果たす。前者をクライアント側、後者をサーバ側として区別する。また、この実装でリソースエージェントは、クライアント側のセンサエージェントのうち、最上位のレイヤに位置するセンサエージェント (Id が 4 のセンサエージェント) から、すべての計算機資源の監視結果を受け取る。クライアント側とサーバ側のセンサエージェントは、相互に通信を行う際、対応するセンサエージェントの持つ Id を算出する必要がある。クライアント側のセンサエージェントが対応するサーバ側のセンサエージェントの Id を算出する場合には式 4.1 を用い、サーバ側からクライアント側への対応は式 4.2 を用いる（式内の ServId はサーバ側のセンサエージェントの Id、ClntId はクライアント側のセンサエージェントの Id、N は各エージェントが生成することのできる子供の数を表す）。

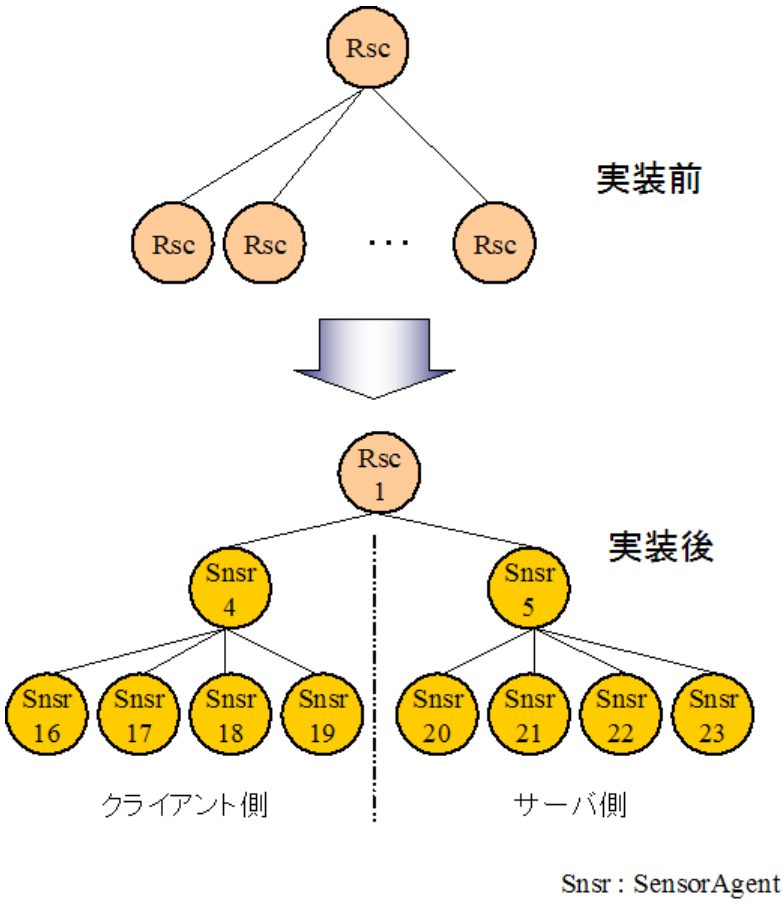

図 4.1.1: センサエージェントの構成

$$ServId = ClntId + N^{\frac{ln(ClntId)}{ln(N)}-1} \tag{4.1}$$

$$ClntId = ServId - N^{\frac{ln(ServId)}{ln(N)}-1} \tag{4.2}$$

葉に位置する各センサエージェントは、監視を開始する前、(1) および (2) の処理を行う。その様子を図 4.1.2 に示す。

(1) クライアント側のセンサエージェント群の根にあたるエージェントは、調査開始時刻の同期をとるために、メッセージを子エージェントに連続して送信する（矢印 1)。このメッセージは、葉に位置するエージェントまで転送される。

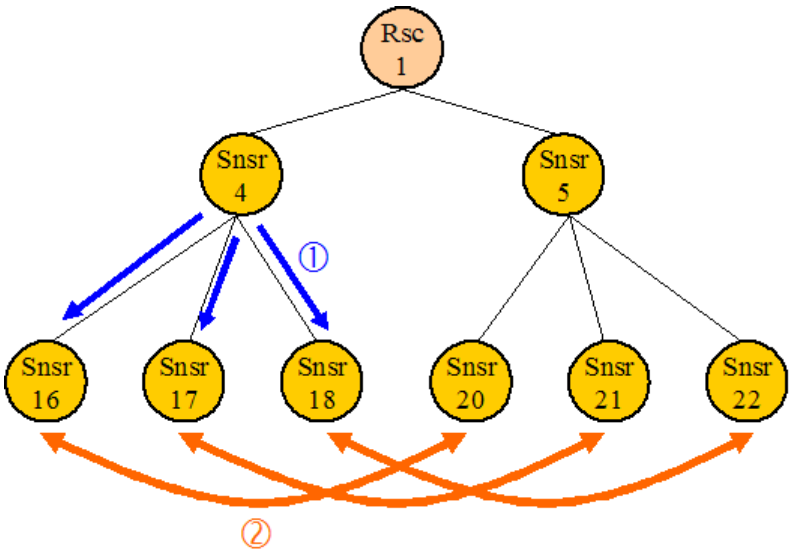

図 4.1.2: (1) および (2) の処理

(2) クライアント側において、葉に位置するセンサエージェントは同期メッセージを受信した後、対応するサーバ側のセンサエージェントに同期メッセージを送る。これを受けて対応するサーバ側のセンサエージェントも、メッセージを返す（矢印 2)。

すべての中間ノードに位置するエージェントは (3)〜(4) の手順を繰り返し実行することにより、計算機資源の監視を行う。その様子を図 4.1.3 に示す。

(3) (2) での同期後、葉に位置するセンサエージェントは最初の調査を開始し、調査結果を親エージェントに送る。それ以降は、前回の調査開始時刻から一定時間経過後に、次の調査を開始する。クライアント側の親センサエージェントは、子エージェントからの結果を受信した後、対応するサーバ側のセンサエージェントに同期メッセージを送り（矢印 3)、資源の調査を開始する。サーバ側の親センサエージェントも、同期メッセージを受信した後、調査を開始する。

(4) サーバ側のセンサエージェントは、資源の調査結果を対応するクライアント側のセンサエージェントに送る（矢印 4)。その結果を受け取ったクライアント側のセンサエージェントは、子供の調査結果を含むすべての結果を親のエージェントへ送る（矢印 5)。

このアルゴリズムは、センサエージェントが所定の計算機で、定期的に監視を行うので、以降、静的監視アルゴリズムと呼ぶ。

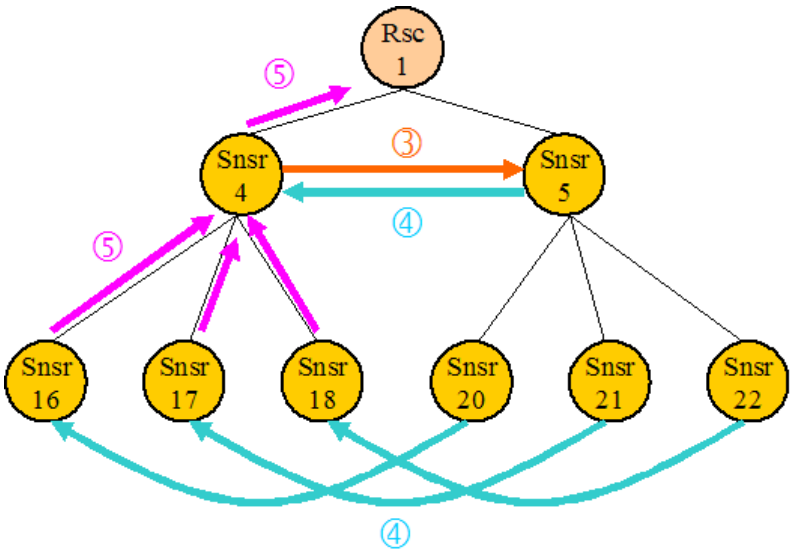

図 4.1.3: (3) および (4) の処理

4.2　動的監視アルゴリズム

静的監視アルゴリズムにおいては、N 台の計算機の監視を行いたい場合に、N 個のセンサエージェントを生成する必要がある。このセンサエージェントに繰り返し計算機間を迂回させることにより、N 台の計算機の監視に対して、N/i 個のセンサエージェントで調査を完了することが可能になる（i は、各センサエージェントが調査する計算機の数)。

このアルゴリズムは、監視対象の各計算機間を移動しながら、資源を調査するため、静的監視アルゴリズムに対し、動的監視アルゴリズムと呼ぶことにする。動的監視アルゴリズムでは、実際の資源の監視は、静的監視アルゴリズムと同様である。したがって、以下では資源調査後の移動に関する手順に注目する。各センサエージェントは、(1)〜(3) の手順を繰り返し実行することにより、計算機を移動する。図 4.2.1 は、その様子を示す。

(1) クライアント側において、葉に位置するセンサエージェントは、次の監視対象の計算機へ移動し（矢印 1)、対応するサーバ側のエージェントに移動完了のメッセージを送る（矢印 2)。サーバ側のセンサエージェントは、そのメッセージを受信後、移動を行い（矢印 3)、同様にクライアント側のセンサエージェントへメッセージを送る（矢印 4)。一方、親のエージェントは、すべての子エージェントから移動が完了したことを知らせるメッセージを受信後、矢印 1 から矢印 4 までと同様の処理を行う。

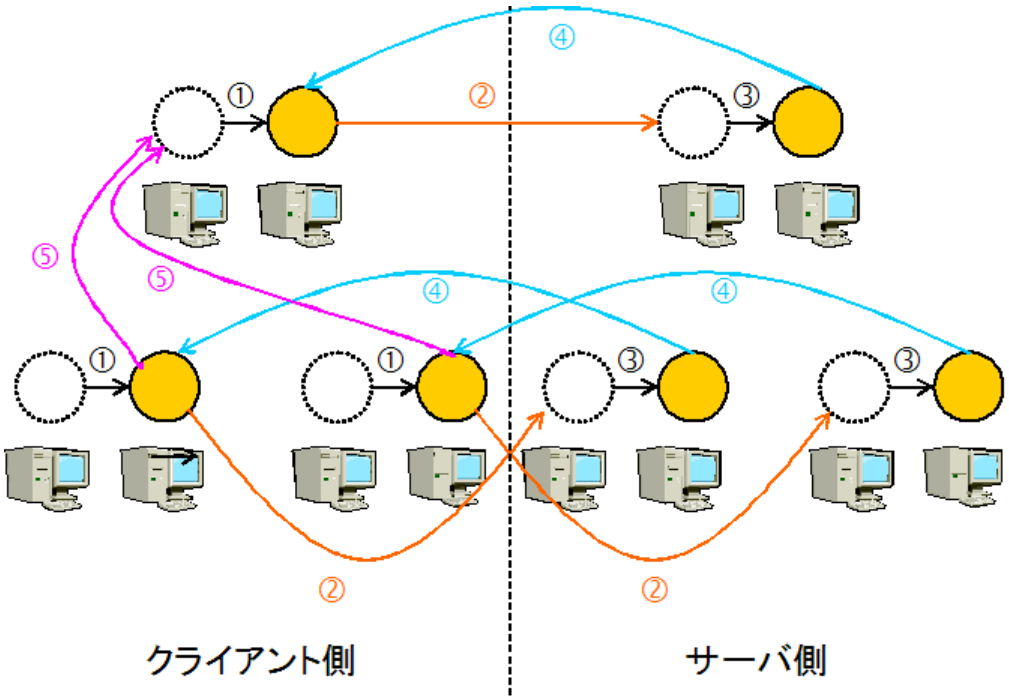

図 4.2.1: エージェントの移動

(2) サーバ側のセンサエージェントからメッセージを受信後、クライアント側のセンサエージェントは、親のエージェントへ移動が完了したことを知らせるメッセージを送る（矢印 5）。

(3) クライアント側のセンサエージェントのうち、すべてのエージェントの根にあたるセンサエージェントが、(2) の処理により送られたメッセージを受信したとき、そのエージェントは、静的監視アルゴリズムに従い、計算機資源の監視を開始する処理を行う。

4.3 測定項目

センサエージェントが測定する資源は以下の通りである（各資源情報の詳しい取得方法は、付録 A.1 から付録 A.5 までに記載する）。

- **CPU の利用可能率** (パーセンテージ)
  CPU の利用可能率は、UNIX コマンドの「uptime」を利用して得られたシステムの実行順番待ちの平均ジョブ数を用いて算出する。ユーザが新たなジョブを一つ追加

するので、この平均ジョブ数に1を加算する必要がある。ユーザプロセスは均等にその占有権が与えられるので、1を加算した平均ジョブ数の逆数をとることにより、各ジョブが均等に使用できるCPUの利用可能率が求められる（式4.3）。

$$available\ CPU\ load\ average = \frac{1.0}{1.0 + UNIX\ load\ average} \times 100 \qquad (4.3)$$

- **OSのタイプ**

  OSのタイプの測定にはJavaのSystemクラスのメソッドであるgetPropertyメソッドを利用する。

- **メモリの空き容量** (メガバイト)

  メモリの空き容量の測定にはUNIXコマンドの「free」を用いる。

- **ディスクの空き容量** (メガバイト)

  ディスクの空き容量を測るためには、UNIXの「df」コマンドを利用して、ユーザアプリケーションに割り当てられた「/tmp」の領域を得る。

- **帯域幅** (Mbps)

  帯域幅の測定には、通信用のTtcpプログラムを使用する。Ttcpは、既にC++で実装されているが、本研究ではJava言語を用いて実装し直した。このプログラムは、ユーザが引数を用いて指定した大きさのデータをクライアントからサーバへ転送し、そのデータサイズと完全に転送されるまでに要した時間から帯域幅を計算する。クライアントとサーバが測定対象とするネットワークは同じであるので、クライアントだけが測定値を得る。クライアント側のセンサエージェントは、Ttcpを用いて帯域幅を測定した後、サーバ側のエージェントから送信されてきた結果にその測定値を加算し、親のエージェントへ送る。

4.4　機能に関する考察

第3.3節に挙げた3つの問題に関して、上述の実装による改善の度合いについて考察する。

(1) 再帰的に子供を生成することのできるセンサエージェントが、計算機資源の情報を取得することで、リソースエージェントはすべての子エージェントと直接通信する必要が無くなり、通信負荷の軽減が期待できる。

(2) 各エージェントが生成できる子供の数が予め n 個までと設定されていて、N 台の計算機資源の情報を知る必要がある場合を想定する（$N \gg n$）。この場合には、N 個のセンサエージェントを再帰的に生成する、あるいは、一つのセンサエージェントに N/n 台の計算機の監視を担当させ、計算機資源の情報の更新頻度が低調にならないようにすることができる。

(3) 帯域幅の測定は対応関係にあるセンサエージェント間で行われるので同じ並列アプリケーションに従事するプロセス間で、通信が必要な場合にも、所定の通信速度を提供できる計算機を、適切に選ぶことができる。

(1)〜(3) の改善点が得られる反面、この実装において、親のセンサエージェントは、子供のエージェントから結果を受け取った後に監視を開始するため、子と親のエージェントの調査開始時刻にずれが発生するが、資源情報の更新頻度を向上するためには、第 4.1 節や第 4.2 節で示したアルゴリズムの実装が必要である。図 4.4.1 は、本アルゴリズムによって、パイプライン方式で資源情報が次々と親エージェントに転送される様子を示す。図中の h はリソースエージェントから葉に位置するセンサエージェントまでの高さを表す。図示したように、すべてのセンサエージェントは、一定の間隔で調査を実施し、結果を親へ送信する。その間隔は、一度の調査に要する時間と親のエージェントが子エージェントの結果を受信するために要する時間の和であり、これは定期的な監視における必要最小限の時間である。ゆえに、資源情報の更新の観点から、本アルゴリズムは、高い更新頻度を維持することができる。

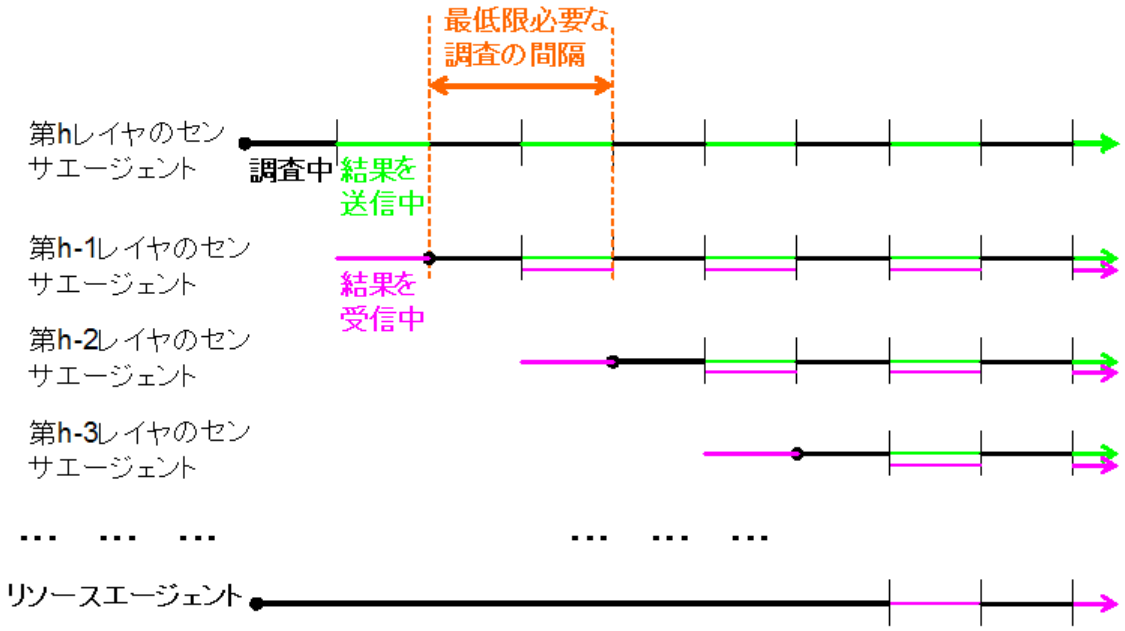

図 4.4.1: 本アルゴリズムにおける資源監視

第5章　性能解析

この章では、センサエージェントと NWS を用いて動的に変化する資源の利用状況を監視し、両者の資源監視能力を比較する実験を行う。はじめに、第 5.1 節で、本実験で計算機資源の利用状況を動的に変化させるために作成したテストプログラムについて説明する。第 5.2 節で、センサエージェントの資源監視における信頼性を検証するため、実際にテストプログラムを計算機上で走らせた状態で、センサエージェントをその計算機に適用して得られた測定結果と、NWS を用いて得られた測定結果について比較する。第 5.3 節では、第 5.2 節でセンサエージェントを用いて資源を調査した際に計測した調査間隔について吟味する。第 5.4 節で、第 5.2 節と第 5.3 節で得られた結果から、センサエージェントの資源監視能力について考察する。

この実験では、バセル校の分散システム研究室で使用されている 32 台の DELL 計算機を用いて構成されたクラスタを利用した。測定は、クラスタゲートウェイ（medusa.uwb.edu）とクラスタノード 8 台（mnode16〜23）を用いて行った（図 5.0.1）。その計算機のスペックを表 5.0.1 に示す。作成したテストプログラムは 5 台の計算機で実行した (CPU 監視用に 1 台、メモリ監視用に 1 台、帯域幅監視用に 2 台、ディスク監視用に 1 台)。5 台の計算機にセンサエージェントを配置するためには、図 5.0.1 に示した木構造を形成する必要がある。ゆえに、mnode16、mnode17、そして mnode23 のクラスタノードにもセンサエージェントは配置されるが、本実験の測定には、参加していない。また、NWS も、同様の 5 台の計算機を使用した。

5.1　テストプログラムの設計

センサーエージェントでのモニタリング機能の動作と正確性を確認するために、異なった計算機上で CPU、メモリ、帯域幅、そしてディスクサイズを一定時間使用し、その後、同じ時間休止する処理を繰り返すプログラムを MPIJava を用いて作成した。表 5.1.1 に、

表 5.0.1: 本研究で使用したワーカーマシンのスペック

CPU	3.2GHz/1MB cache Xeon
OS	Linux
メモリ	512MB
ハードディスク	36GB SCSI
ネットワーク	1Gbps(Giga Ethernet)

本テストプログラムのテスト項目の詳細を示す。

表 5.1.1: テストプログラム実行時の処理

計算機	測定項目	処理
mnode18	CPU の負荷	ハノイの塔の計算を繰り返す
mnode19	メモリサイズ	20Mbyte の配列を確保する。ただし、処理の休止時に開放されるメモリのサイズは、JVM のガーベジコレクトに依存するため、確保したサイズ分のメモリが完全に開放されるとは限らない。
mnode20、mnode21	帯域幅	4M バイトのデータを送受信し続ける
mnode22	ディスクサイズ	50Mbyte のファイルを「/tmp」ディレクトリに作成する

5.2　監視機能の信頼性

本実験では、テストプログラムの実行時間を 10、20、80 秒の 3 つの場合において実行した。また、計算機資源を監視する間隔は、子エージェントの測定開始時刻から親のエージェントが全ての子エージェントの結果を受け取るまでにかかる時間を考慮して決めた（ただし、現在の AgentTeamwork の設定にもとづき、各エージェントが生成できる子エージェントの数を 4 として考える）。その時間の導出方法は次の通りである。

はじめに、資源の測定にかかる時間（クライアント側のエージェントが調査を開始してか

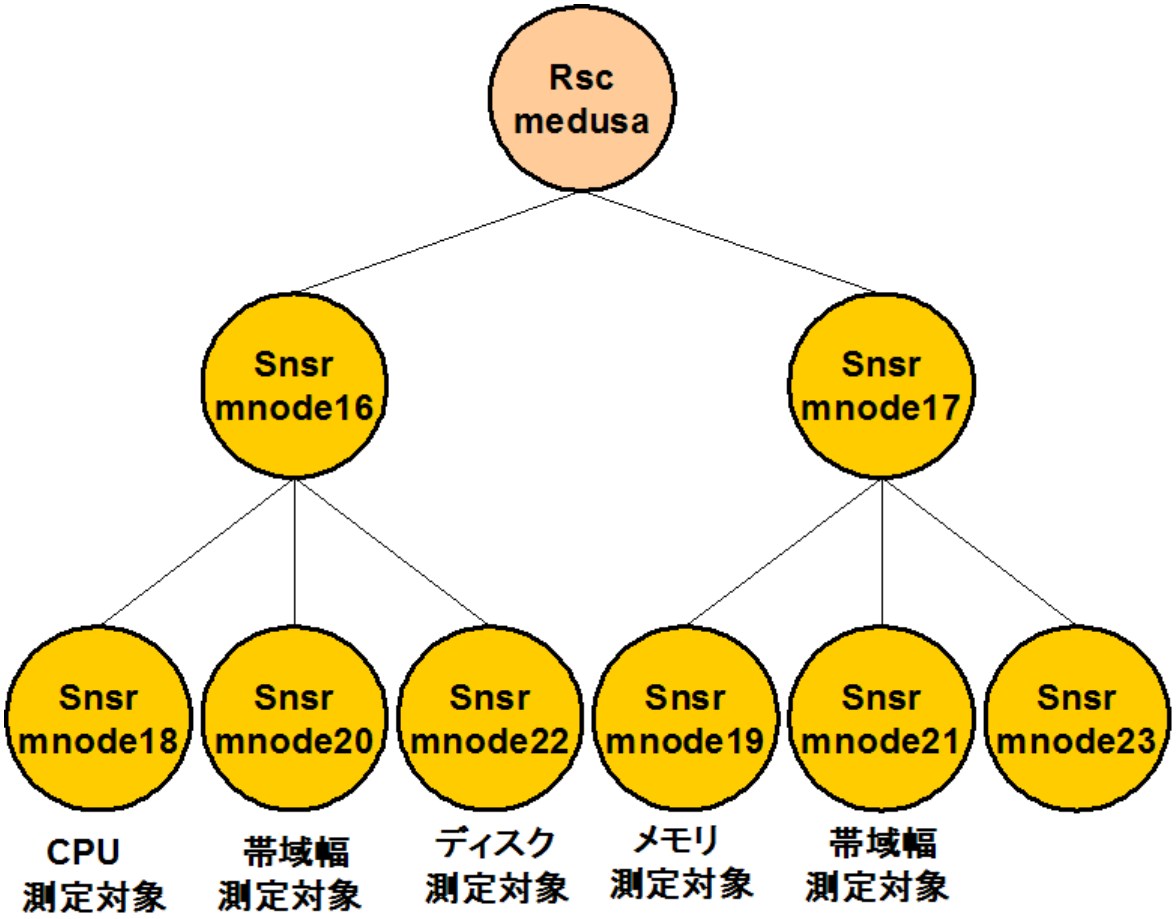

図 5.0.1: 実験時のセンサエージェントの構成

ら結果を親のエージェントに送信する直前の時刻までの時間）を図 5.2.1 を用いて考える。

- CPU、メモリ、ディスクサイズ、OS のタイプを調査するためにかかる時間は、数ミリ秒から十数ミリ秒程度である
- クライアント側のセンサエージェントがサーバ側のエージェントの結果を受け取るためには、数十ミリ秒から 3 秒程度の時間がかかる。現在の AgentTeamwork において、他のエージェントとの通信には RMI を用いており、その処理方法は RMI の実装に依存している。ゆえに、この時間は、試験的に RMI での通信にかかった時間を計測して、結果として得た値である。
- 帯域幅の測定には、2 秒程度の時間がかかる。帯域幅を測定するのはクライアント側のエージェントのみであるので、サーバ側のエージェントの測定時間には含まれない。ただし、測定時のネットワークの状態により、この時間は増減する。

以上より、本クラスタ上でセンサエージェントが資源の測定にかける時間は、5 秒程度である。

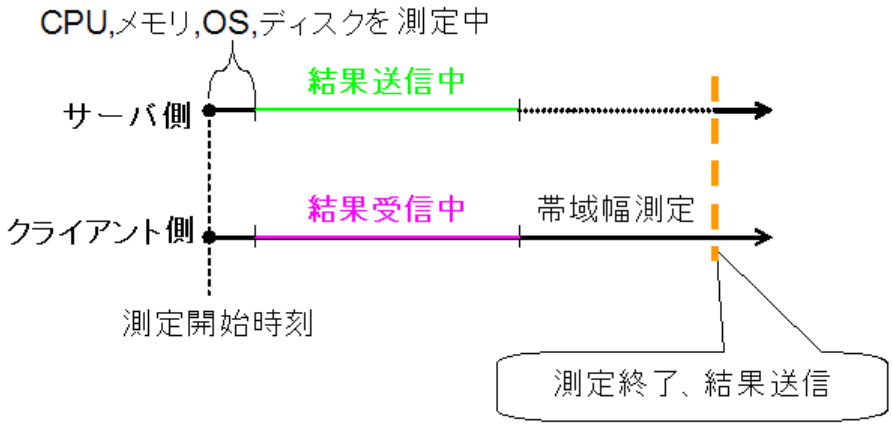

図 5.2.1: 一度の測定にかかる時間

葉に位置する子エージェントが、親のエージェントへ結果を送る際にかかる時間を図 5.2.2 を用いて考える。各エージェントが親のエージェントへ結果を送信する際には、最大で 3 秒程の時間がかかる。これは、上述したようにエージェント間の通信が RMI の実装に依存するからである。各エージェントが RMI を呼び出した際、その要求はキュー内に容れられるので、通信は逐次的に処理される。各通信に最大 3 秒かかると仮定した場合に、親が子供のエージェントの結果を完全に受信するまでに、最大で 12 秒かかる。これらの要因から、センサエージェントは、17 秒程度の間隔を空けて、測定を行わなければならない。一方、NWS の場合は、同じ資源を同じ条件で測定するために、最短で 4 秒程度かかる。以上の点から、測定の間隔は、帯域幅測定にかかる時間や通信時間の増減を考慮して、20 秒とした（もちろん、この測定間隔は、異なるクラスタおよびクラスタ間では増減する）。

以上より算出した間隔で監視を行い、得られた結果を CPU（図 5.2.3 から図 5.2.5）、メモリ（図 5.2.6 から図 5.2.8）、帯域幅（図 5.2.9 から図 5.2.11）、ディスク（図 5.2.12 から図 5.2.14）の順で記載する（各図題の t はテストプログラムの実行間隔を秒で表している）。帯域幅は、mnode20 と mnode21 で測定したが、監視対象のネットワークは同じであるため、mnode20 における帯域幅の調査結果を記載する。また、監視間隔が 20 秒であるので、テストプログラムの実行時間が 10 秒の場合は、常にテストプログラム実行時の資源が測定される。20 秒の場合は、テストプログラム実行中および休止中の資源を交互に測定し、80

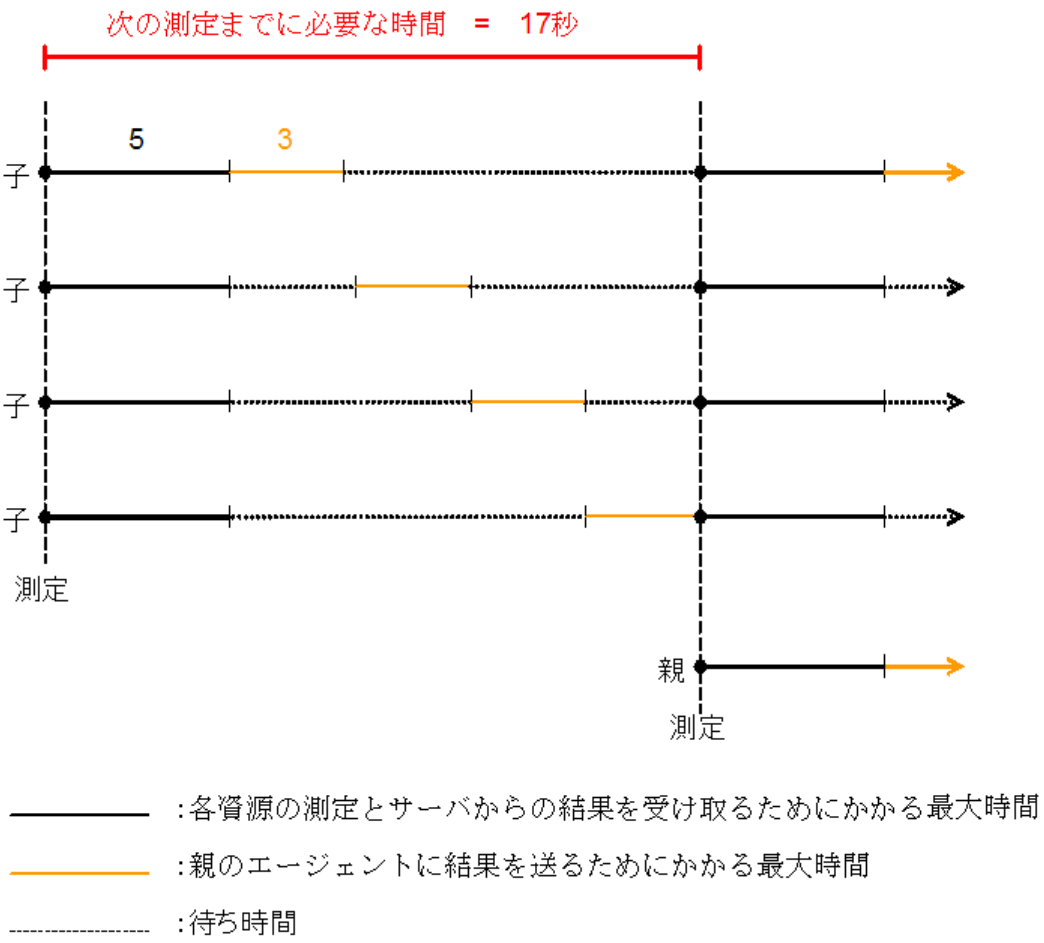

図 5.2.2: 最大必要な測定間隔

秒の場合は、テストプログラム実行中および休止中に 4 度ずつ交互に測定が行われる。これらをもとに、センサエージェントと NWS による各計算機資源の測定値を比較する。

- **CPU**

  NWS は、CPU の利用可能率を算出するために、uptime および vmstat コマンドと、さらにコマンドを使わない独自の方法を用いる。uptime および vmstat を用いるのは、コマンドの実行による CPU への負荷が少ないためである。また、独自の方法は、CPU を多く使うプログラムを走らせ、実際の実行時間と CPU が使用された時間の比から、CPU の利用可能率を算出する方法であり、コマンドよりも正確に利用可能率を算出できる。ただし、独自の方法を用いると、CPU 自体に負荷がかかるため、頻繁には使用されない。そこで、NWS は通常、独自の方法により算出された値と uptime および vmstat によって算出された値をそれぞれ比較して、前者の値と誤差の少ない

方のコマンドを測定に利用し、そのコマンドにより算出された値に、誤差を加減した値を測定値とする。これにもとづいて、両者のグラフを比較する。図 5.2.3 に示すように、テストプログラムの実行間隔が 10 秒の場合、両者の測定値は、ほぼ同様のグラフを描いている。実行間隔が 20 秒の場合（図 5.2.4)、テストプログラム実行時には、NWS の方がより期待値に近い値を示しているが、休止時にはセンサエージェントの方が期待値に近い値を示しているため、一概にどちらの測定が優れているとは言えない。しかしながら、実行間隔が 80 秒の場合には（図 5.2.5)、NWS の方が、期待値に近い値を示すことから、NWS の測定の方が正確である。これは、独自の方法と uptime による測定値の誤差、および、独自の方法と vmstat による測定値の誤差を比較した際、どちらの誤差の値が小さいかによって、実行されるコマンドが入れ替わっている様子を示す。例えば、100 秒の時点では、独自の方法により算出された値と uptime および vmstat により算出された値の誤差を比較した際、uptime との誤差が少なかったため、150 秒前後に独自の方法が実行されるまで、uptime を利用して測定が行われている。しかしながら、150 秒前後で、同じように誤差を比較した際、今度は vmstat との誤差が少なかったため、250 秒前後に独自の方法が実行されるまでは、vmstat が利用されている。

- **メモリ**

  図 5.2.6 から図 5.2.8 が示すように、両者とも、すべての場合において同程度の測定値を得ることができている。テストプログラム実行中に 20M バイトのメモリを確保したにもかかわらず、休止時と実行時と測定値の差異が 10M バイト程度であるのは、JVM のガーベジコレクトにより確保したメモリが完全に開放されなかったためである。

- **帯域幅**

  図 5.2.10 および図 5.2.11 において、テストプログラム休止時には両者とも、同程度の測定値を示している。しかしながら、図 5.2.9 から図 5.2.11 に示すように、実行時には、センサエージェントが 600Mbps 前後の測定値を示しているのに対して、NWS は 400Mbps から 500Mbps ほどの測定値を示す。NWS の方が、より期待値に近い測定値を示しており、測定精度は NWS の方が優れている。

- **ディスク**

  図 5.2.12 から図 5.2.14 ディスクサイズの測定値は、両者とも同様の結果が得られて

いる。また、その測定値は、休止時が 920M バイトで、実行時が 870M バイトであることがから、結果も正しいことが分かる。

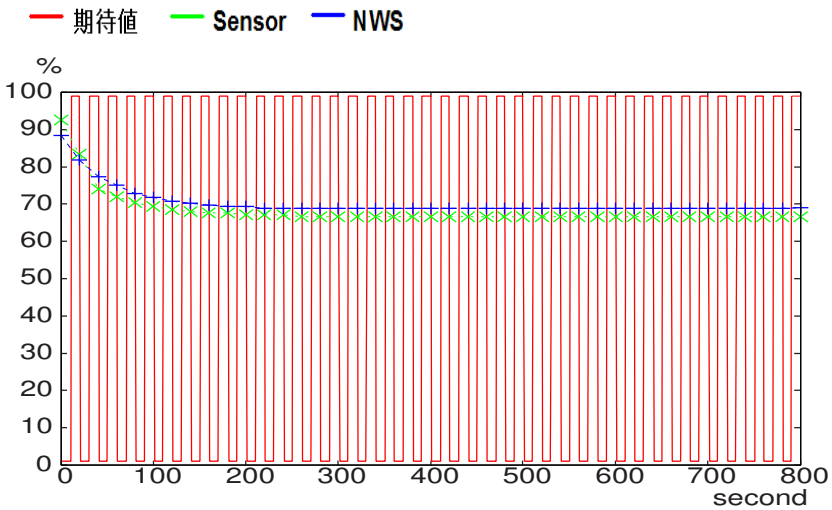

図 5.2.3: t=10 秒での CPU の利用可能率の測定

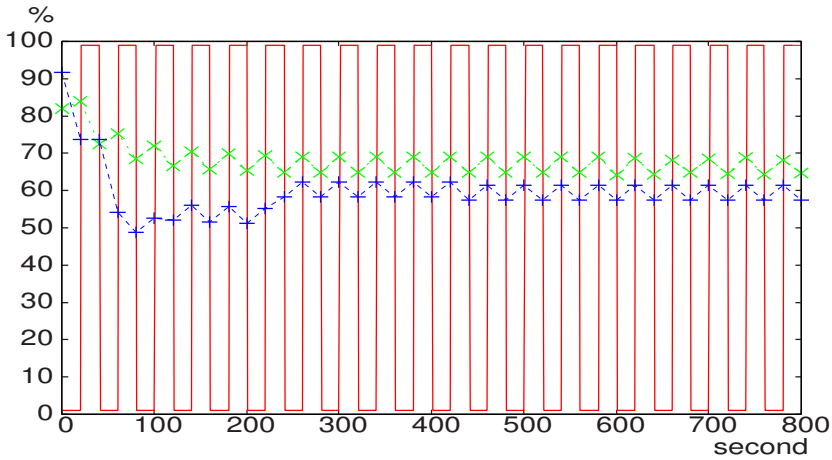

図 5.2.4: t=20 秒での CPU の利用可能率の測定

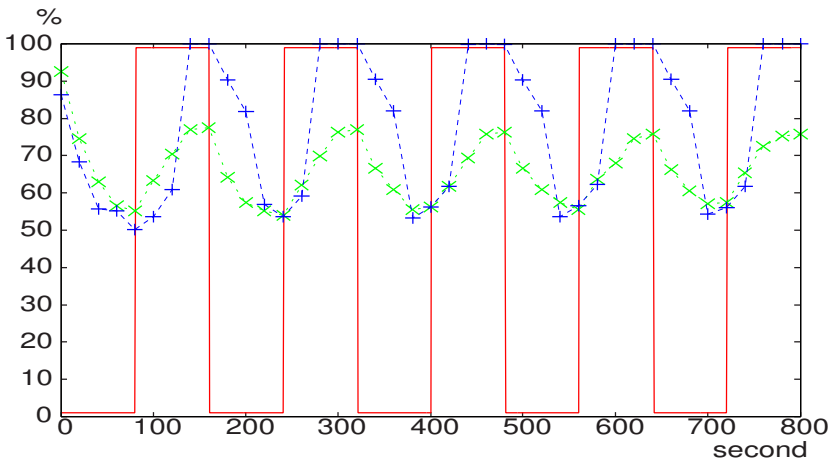

図 5.2.5: t=80 秒での CPU の利用可能率の測定

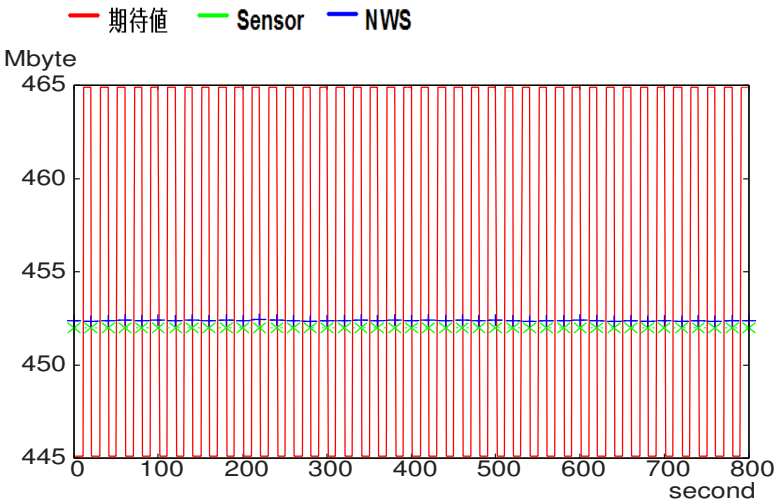

図 5.2.6: t=10 秒でのメモリの測定

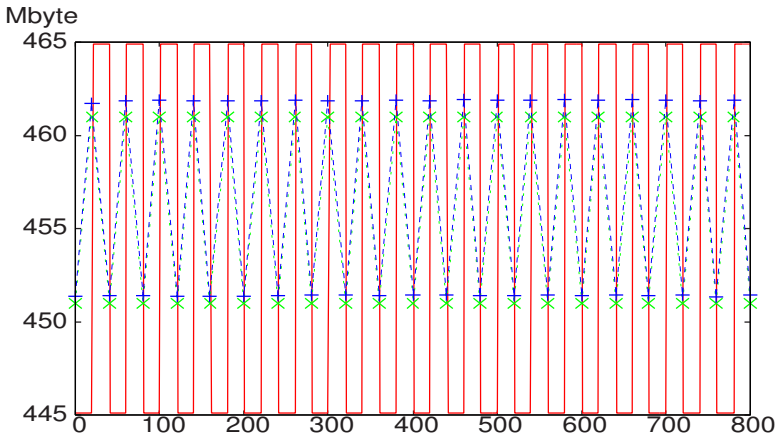

図 5.2.7: t=20 秒でのメモリの測定

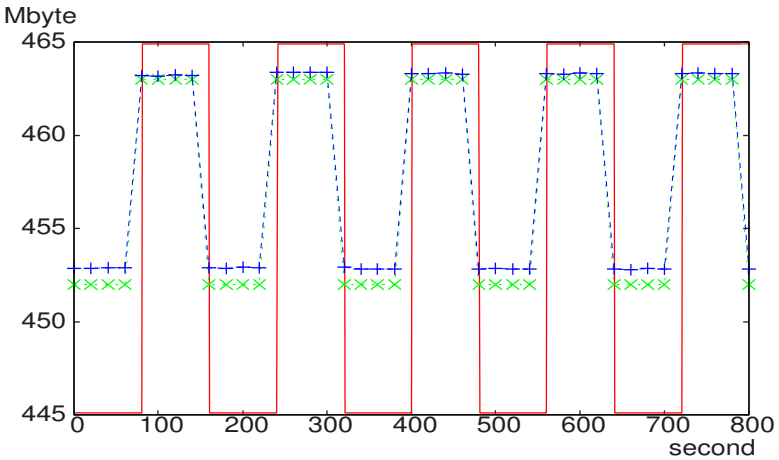

図 5.2.8: t=80 秒でのメモリの測定

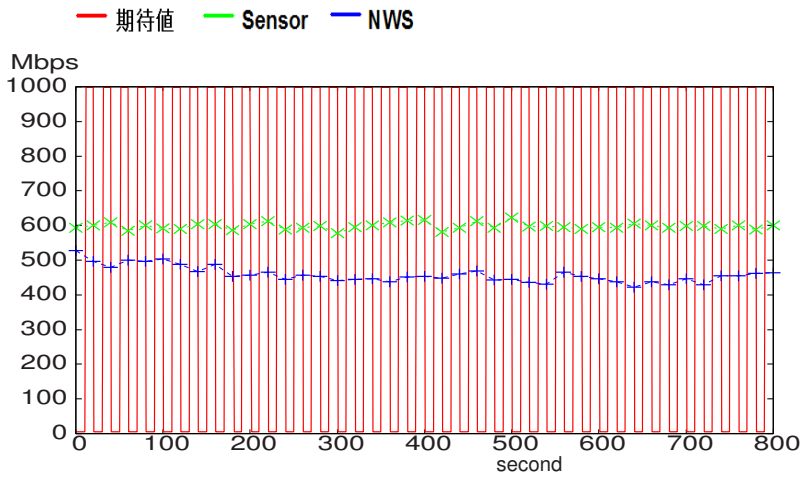

図 5.2.9: t=10 秒での帯域幅の測定

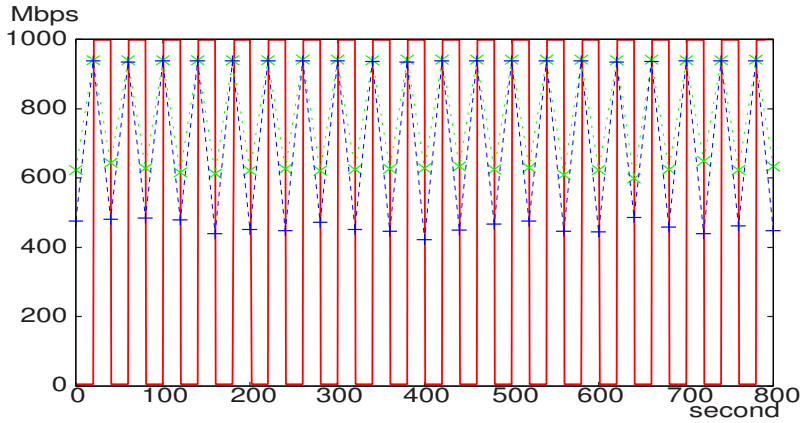

図 5.2.10: t=20 秒での帯域幅の測定

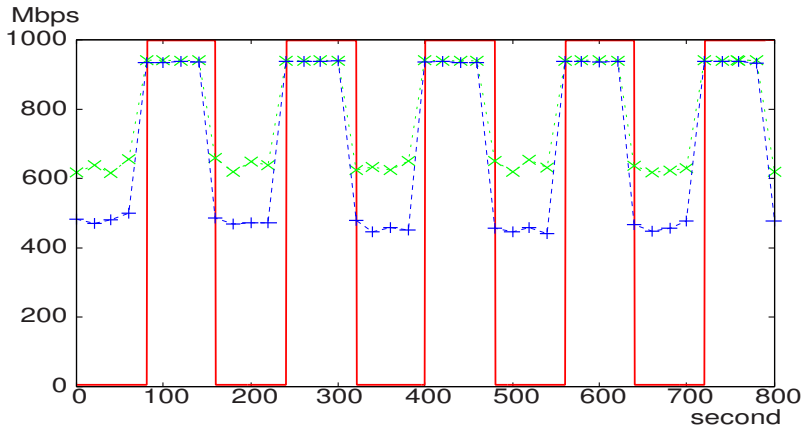

図 5.2.11: t=80 秒での帯域幅の測定

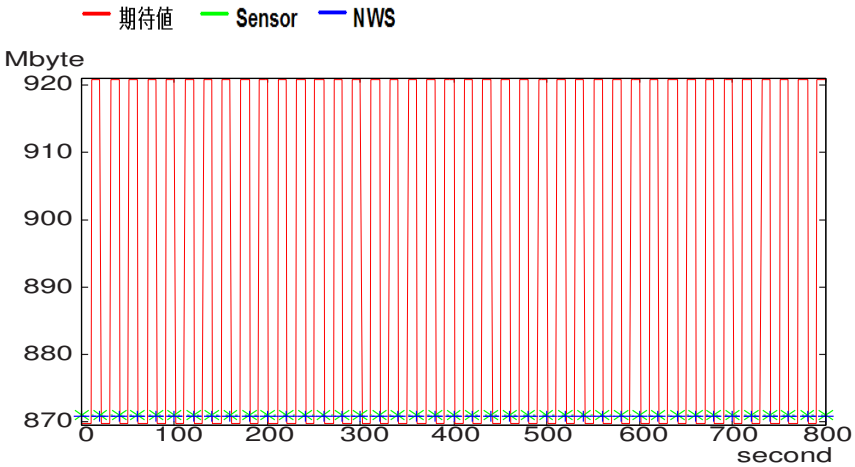

図 5.2.12: t=10 秒でのディスクサイズの測定

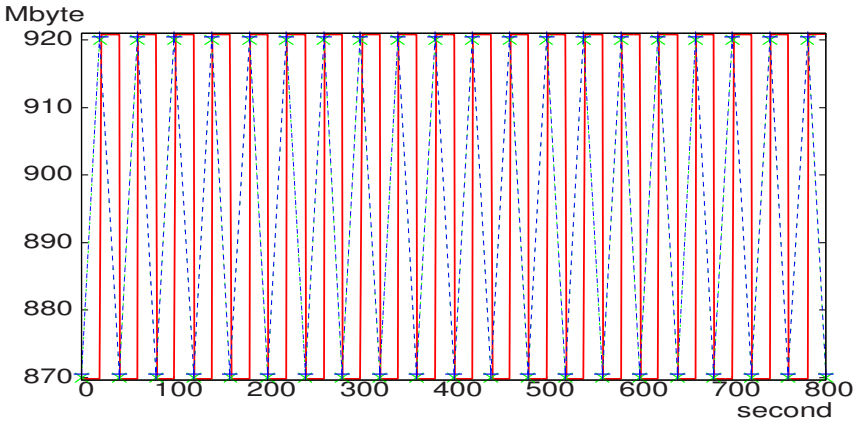

図 5.2.13: t=20 秒でのディスクサイズの測定

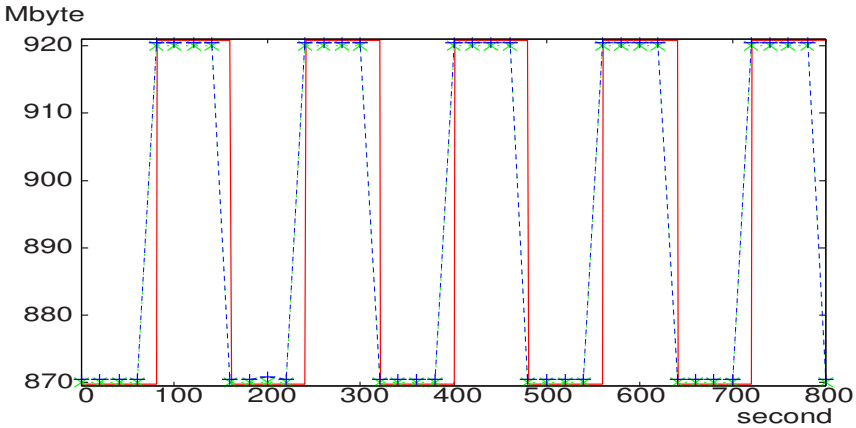

図 5.2.14: t=80 秒でのディスクサイズの測定

5.3 測定性能

本節では、資源の監視を行った際に、センサエージェントが要した実際の測定時間について調査する。図5.3.1に、mnode18およびmnode20上の各センサエージェントが、資源の調査に要した時間を示し（mnode22上のセンサエージェントからは、mnode18上のエージェントと同様の結果が得られているので省略する）、図5.3.2に、各センサエージェントが親のエージェントに結果を送信するために要した時間を示す。

mnode18上のセンサエージェントが、資源の測定に要した時間は1800ミリ秒程度であり、サーバ側のエージェントから結果を受信するために要した時間も数ミリ秒から数十ミリ秒程であるため、資源の調査に要した時間は2秒にも満たない。しかしながら、mnode20上のセンサエージェントが調査に要した時間は最大で3秒ほどである。mnode20では、MPIJavaのテストプログラムの実行でネットワークが混雑しており、帯域幅の測定に時間がかかったからである。この場合も、サーバ側のエージェントから結果を受信するために要した時間は、前者と同程度である。

異なる計算機上の各エージェントが、結果を親のエージェントへ送信するために要した時間にも、極端な変動がある。mnode18およびmnode20上のエージェントは、その通信に1秒も必要としていないが、mnode22上のエージェントは最大で3秒程度かかっている。これは、第5.3節に示したように、子エージェントが親のエージェントに結果を送信する際に、RMIによる通信時間の程度の違いが顕著に現れている状況を示している。これらの結果から、mnode18〜mnode23で測定が開始されて、親のエージェントが結果を受信し終えるまでに要した最大時間は、6秒程度である。ゆえに、20秒の間隔は、監視の間隔として十分であることが実証された。

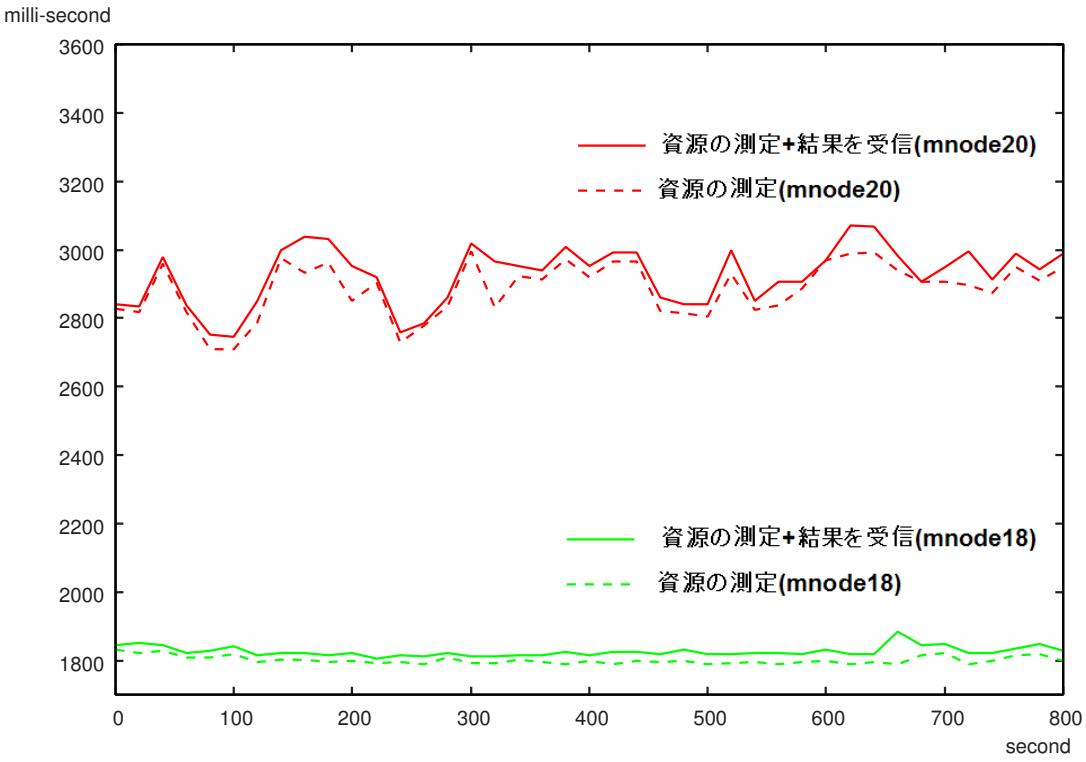

図 5.3.1: 資源の調査に要した時間

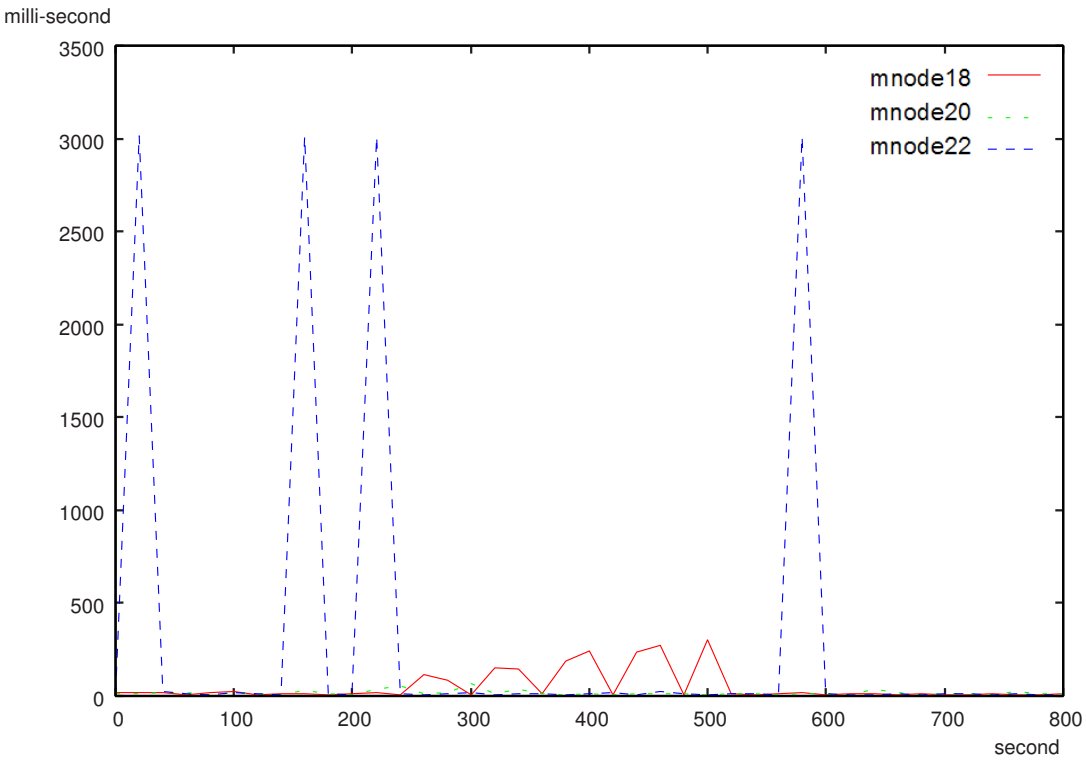

図 5.3.2: 結果の送信に要した時間

5.4　考察

第 5.2 節より得られた結果を概括する。

- **CPU**
  実行間隔が 10 秒および 20 秒の場合、両者の測定の是非を判断することはできないが、80 秒の実行間隔の場合、NWS の測定値の方が正確であった。ゆえに、本実験の場合、テストプログラムの実行間隔が長くなるほど、NWS の方が正確な値を測定できる。

- **メモリ**
  両者とも同程度の値を示す。

- **帯域幅**
  テストプログラムが休止時には、同程度の測定値を得ることができているが、実行時には、NWS の方がより期待値に近い値を示している。

- **ディスク**
  両者とも同程度の値を示す。

以上の結果から、メモリおよびディスクサイズの測定には、センサエージェントは NWS と遜色ない結果を得ることが可能であると言える。しかしながら、CPU および帯域幅の測定値は、NWS の方がより正確な値を示した。AgentTeamwork および NWS は、長期的に運用することを目的としたシステムであるため、監視の間隔を長くすることも可能であるが、その場合、やはり正確な CPU 利用可能率を知る必要がある。以上より、CPU および帯域幅を、さらに正確に測定する必要がある場合には、NWS の測定方法をセンサエージェントに適用する、あるいは、ユーザの選択によって、センサエージェントから NWS を起動できるようにする必要がある。

第 5.3 節に示した結果より、見積もった測定間隔は正しいことが立証されたが、監視の間隔が最短で 4 秒である NWS に比べて、資源情報の測定間隔が長いことが分かった。この問題点を解決するためには、エージェント間の通信に要する時間を、短縮する必要がある。現在、AgentTeamwork のプロジェクトでは、RMI を利用しての通信をソケット通信へと転換する取り組みが実施されている。これにより、RMI よりも高速な通信が期待されるため、この実装後に改めて実際の測定間隔を調査する必要がある。

第6章　結論

本研究で行われたAgentTeamworkシステムの資源監視アルゴリズムの改善、および、既存の資源監視システムとの比較により得られた成果、および、今後の改善点について概括する。

成果

- 資源監視を行うセンサエージェントを再帰的に生成することで、リソースエージェントにかかる負荷を減少させることができた。
- センサエージェントが調査対象の計算機間を移動することで、資源情報の更新頻度の低下を抑制することができた。
- 対応するセンサエージェント間のネットワークの帯域幅を測定することで、並列アプリケーションのプロセス間通信の性能向上を図ることができた。
- メモリおよびディスクサイズをNWSと同等の精度で測定することができた。

今後の改善点

- CPUおよび帯域幅の測定には、NWSほどの精度が得られなかった。ゆえに、これら2つの資源をより正確に測定する必要がある場合には、NWSの測定方法をセンサエージェントに適用する、あるいは、センサエージェントからNWSを起動できるようにする必要がある。
- センサエージェントの測定間隔をNWSよりも短くする必要がある場合には、エージェント間の通信時間を短縮しなければならない。現在、AgentTeamworkシステムでは、RMIによる通信をソケット通信に移行しており、その後は、さらに高速な通信が期待されるので、改めて通信時間を調査する必要がある。

謝辞

AgentTeamwork は、米国立科学財団 (National Science Foundation) Shared Cyberinfrastructure プログラムの支援（登録番号 0438193）を受けて、ワシントン大学で遂行されているプロジェクトであり、本卒業研究は、AgentTeamwork プロジェクトの一環として、ワシントン大学バセル校および愛媛大学との間の学術協定にもとづく交換留学制度のもとで、実施させていただきました。社会の発展に貢献する貴重な研究に参加させていただいたことに感謝します。

卒業論文作成にあたり、ご指導を受け賜わった小林真也教授に心から感謝します。また、卒業論文だけでなく、ワシントン大学バセル校と愛媛大学との交換留学という形でアメリカワシントン州での修学という貴重な体験をすることができ、その機会を与えていただいたことにも深く感謝します。

柏木紘一助手には、留学前に何度か本研究に関してのゼミを開いていただき、プロジェクト参加の事前に、ある程度の知識の獲得ができたことに、真に感謝します。

留学中は、ワシントン大学バセル校で教鞭をとられておられる福田宗弘助教授に、終始ご懇切なるご指導ご鞭撻を受け賜わり真に感謝します。今回の卒業論文のテーマとして福田助教授が率いる AgentTeamwork のプロジェクトに参加することができ、多謝の念を感じます。私にとっては、難しい研究内容でしたが、福田助教授には忍耐強く教育を施していただきながら、本研究をやり遂げられましたことに心から感謝します。

また、海外に不慣れな私を暖かく迎えて、生活面でたくさんの協力をしていただいたホストファミリーの皆様のおかげで、海外生活に不自由することなく本研究をやり遂げられたことに大いに感謝します。

今回はワシントン大学バセル校で講義をいくつか受講させていただきました。講義を受けて、自分に不足していた知識を補うことができ、本研究の完遂のために役立てることができました。受講の機会を与えてくださった、計算ソフトウェアシステム学科ジャクル学科長、メスキ女史、ハンタ女史に感謝の意を表します。

本研究はワシントン大学バセル校を卒業されたマック氏の研究の内容の一部を引き継ぐもので、研究に携わり始めた当初、マック氏に、研究内容を解説していただき、本研究に支障なく取り組むことができました。大変感謝します。

参考文献

[1] Milojicic, D. et al., *"Mobility Processes, Computers, and Agents"*, Addison-Wesley, 1999

[2] Munehiro Fukuda, Koichi Kashiwagi and Shinya Kobayashi,
"AgentTeamwork: Coordinating Grid-Computing Jobs with Mobile Agents", In Special Issue on Agent-Based Grid Computing, International Journal of Applied Intelligence, in 2006

[3] Enock Mak and Munehiro Fukuda,
"A Development of Resource/Commander Agents Used in AgentTeamwork Grid Computing Middleware",
http://depts.washington.edu/dslab/AgentTeamwork/report/enoch_sp05.pdf, March 17 2005

[4] Enock Mak and Munehiro Fukuda,
"Development of Resource/Commander Agents For AgentTeamwork Grid Computing Middleware",
http://depts.washington.edu/dslab/AgentTeamwork/report/enoch_wi05.pdf, June 20 2005

[5] NETWORK WEATHER SERVICE, http://nws.cs.ucsb.edu/ , September 16 2005

[6] NSF MIDDLEWARE INISIATIVE - Network Weather Service,
http://archive.nsf-middleware.org/documentation/NMI-R5/0/gridscenter/NWS/,
August 2004

[7] NWS のインストールおよび環境の構築方法,
http://mikilab.doshisha.ac.jp/dia/research/report/2004/0612/004/report20040612004.html,
December 10 2004

[8] A Worldwide Virtual Computer, http://www.cs.virginia.edu/ legion/ ,2001

[9] Andrew S. Grimshaw,Anand Natrajanm,Marty A. Humphrey,Michael J. Lewis,Anh Nguyen-Tuong,John F. Karpovich,Mark M. Morgan and Adam J. Ferrari, *"From Legion to Avaki: The Persistence of Vision"* , Grid Computing: Making the Global Infrastructure a Reality, F. Berman, G. Fox and T. Hey, Eds., John Wiley & Sons, Hoboken, NJ, 2002

[10] Brian S. White,Andrew S. Grimshaw and Anh Nguyen-Tuong, *"Grid-Based File Access: The Legion I/O Model"*, Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing, Pittsburgh, August 2000

[11] Peter J. Keleher,Jeffrey K. Hollingsworth and Dejan Perkovic, *"Exposing Application Alternatives"* ,ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, page 384, IEEE Computer Society, 1999

[12] Steve J. Chapin,Dimitrios Katramatos,John Karpovich and Andrew Grimshaw, *"Resource Management in Legion"* ,Future Gener. Comput. Syst., Vol.15, page 583-594, http://dx.doi.org/10.1016/S0167-739X(99)00011-4, Elsevier Science Publishers B. V., December 3 1997

[13] Legion 1.8 Reference Manual, http://legion.virginia.edu/documentation/tutorials/1.8/manuals/Reference_1_8.html, 2001

[14] Condor Project Homepage, http://www.cs.wisc.edu/condor/ , June 22 2005

[15] Optena's Condor Knowledge Base, http://docs.optena.com/display/CONDOR/Optena's+Condor+Knowledge+Base , December 17 2005

[16] Setting up Condor for Special Environments, http://www.cs.wisc.edu/condor/manual/v6.3 , 2001

[17] The MW Homepage, http://www.cs.wisc.edu/condor/mw/ , 2001

[18] Condor Version 6.6.10 Manual, http://www.cs.wisc.edu/condor/manual/v6.6/ref.html , 2001

[19] Rich Wolski, Neil Spring and Jim Hayes, *"Predicting the CPU Availability of Time-Shared Unix Systems on the Computational Grid*, HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing, page 12, IEEE Computer Society, 2000

付 録A　各計算機資源の測定手段

第 4.3 節に記載した各資源の測定に用いた方法を記す。

A.1　CPU

uptime コマンドを使用すると、次のような出力を得ることができる。

```
07:32:42 up  4:15,  1 users,  load average: 0.21, 0.10, 0.00
```

注目すべき点は、load average より右の 3 つの数字である。それぞれ、過去 1 分間、5 分間、15 分間において、システムの実行順番待ちの平均ジョブ数を表す。このうち、過去 1 分間の値、すなわち一番目の値を用いて、CPU の利用可能率を算出する。(上記の場合では、0.21 である)。[19]

A.2　OS のタイプ

OS のタイプを取得するために、コード内では、次のように実行する。

```
    System.getProperty( "os.name" );
```

このメソッドは引数により、目的の値を取得できる。センサエージェントは OS のタイプを計算機の情報として知る必要があるので、引数を「os.name」として、OS のタイプを取得する。

A.3　メモリ

free コマンドを実行すると次のような出力が得られる（この出力における値はすべてキロバイト単位である）。

```
             total       used       free     shared    buffers     cached
Mem:        515548     362288     153260          0      99816     187352
-/+ buffers/cache:      75120     440428
Swap:      1052216          0    1052216
```

実際の空き容量は、「Mem」の行の「free」列の値 153260 ではなく、その下の 440428 である。「Mem」の行の「free」列に記されている値は、実際の空き容量からバッファとキャッシュのために確保されている分を引いた値である。バッファとキャッシュのために確保されている値は、「Mem」行にそれぞれ表記されている。ゆえに、これらの値を加算した値が実際に利用可能な空き容量となる。「Mem」行の free、buffers、cashed の値を加算すると、その合計値は 440428 になることが分かる。

A.4　ディスク

ディスクサイズを得るために、コード内では、「df -m /tmp」というコマンドを実行して値を取得している。第 1 引数の「-m」は空き容量をメガバイト単位で返すためのオプションである。第 2 引数の「/tmp」は、空き容量を知りたいディレクトリのパスを指定している。

A.5　帯域幅

表 A.5.1 は Ttcp プログラムで使用する引数の一覧である。ただし、ここに挙げた引数は、センサエージェントが実際に使用するオプションとセンサエージェントのために追加したオプションである。

センサエージェントがリソースエージェントから生成される際、一意のポート番号がセ

表 A.5.1: Ttcp 利用の際のオプション一覧

オプション	機能
-r	Ttcp 通信におけるサーバを指す（サーバ側のセンサエージェントが指定）
-t	Ttcp 通信におけるクライアントを指す（クライアント側のセンサエージェントが指定）
-l#	ネットワークから読み込まれる、または書き込まれるバッファのサイズを表す (デフォルトは 8192、単位はバイト)
-b#	ソケットバッファのサイズ (デフォルトは 16384、単位はバイト)
-p#	通信で使用するポート番号
-i#	エージェントの Id（クライアントの場合は、対応するサーバ側のエージェントの Id であり、サーバ側のエージェントの場合は自身の Id である。）
-n#	ネットワークへ書き込まれるバッファの数
-m#	サーバをマルチスレッドにするかどうか (0 の場合マルチスレッド、1 の場合はマルチスレッドでない。サーバ側のエージェントのみ使用)

ンサエージェントのコンストラクタの引数として渡され、すべてのセンサエージェントはそのポート番号を用いる。このポート番号は、リソースエージェントが生成された時に発行されたタイムスタンプを 56536 で割った剰余に 10000 を加算した値である。Id は Ttcp のサーバを使用する際の認証に用いられる。Ttcp のサーバに、他のユーザがアクセスしてきた場合に、Id を用いた認証を施すことで、そのユーザがサーバを無断で利用できないようにする。Ttcp のクライアントにあたるセンサエージェントが、対応するサーバのセンサエージェントの Id を、サーバへ送信し、サーバ側のエージェントは、それを受け取り自らの Id と一致していれば、データの転送を許可する。サーバをマルチスレッド化するかどうかは、使用するアルゴリズムによって決められる。静的監視アルゴリズム用いて、資源を監視する場合は、マルチスレッドとしてサーバを起動する。このアルゴリズムでは、定期的に資源の調査が行われるため、調査の度にサーバの起動とシャットダウンを繰り返すの

は、効率的ではない。そこで、マルチスレッドを適用し、サーバを常に起動しておく。一方、動的監視アルゴリズムの場合は、一度測定を終えるとセンサエージェントは次の計算機へ移動するので、サーバは各計算機での測定を終える度にシャットダウンされる。