修 士 論 文

ディスコネクティッド オペレーションをサポート する 適応可能なアプリケーションの構築

指導教官　中島達夫 助教授

北陸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

嶋本 堅司

1998 年 2 月 13 日

要 旨

本論文では、ユーザが各メトリクッス間のトレードオフを考慮・定義できる汎用的なインタフェースを提供するとともに、それらを複数の異なる実装へマッピングするためのフレームワークやメカニズムを提供する。

ユーザはこのインタフェースを用いて、考慮すべきメトリック種別とそのメトリックスの重要度を表す数値パラメタを指定する。インタフェースと実装をマッピングするために用意されたマッピングレイヤーが、指定されたパラメタを計算して実装種別を決定する。マッピングレイヤーと実装の間にもインタフェースが定義されており、マッピングレイヤーがこのインタフェースを通して、実装レイヤーがサポートしている実装種別を呼び出す。両インタフェースが定義されているため、これらのインタフェース仕様に適合する限り、マッピングレイヤーの内部を簡単に変更することができる。

これにより、ユーザやアプリケーションの要求に適応可能なアプリケーションを構築することが可能となる。

実際の検証作業としては、まずオブジェクト指向のアプリケーションに対してディスコネクティッドオペレーションの機能を提供するツールキットを用意した。このツールキットは、通信コストと共有データの一貫性のトレードオフに基づいた異なる実装をサポートしている。そして、アプリケーションの例として少人数で使用するグループスケジューラを作成し、実際の用途をシミュレーションすることにより、通信コストと共有データの一貫性のトレードオフをユーザが考慮することの有効性を実証した。

ツールキットの作成では、これまでほとんど重要視されてこなかった楽観的並行制御における競合解消について、様々な競合解消戦略を提案し、ユーザの要求に基づく一貫性維持の実現を目指した。さらに、従来の研究では全く指摘されてこなかったアプリケーションのセマンティクスへの依存性や、競合解消に伴なうシリアライザビリティの保証に関する問題点についても提起する。

目 次

第 1 章

はじめに

現在、電子装置の小型化、高性能化、低消費電力化等の技術進歩により、移動計算機上での様々なアプリケーションの利用が可能になってきている。また、携帯電話や PHS 等の無線を利用した通信メディアの普及や、PIAFS 等に見られる通信回線の高速化により、移動先での既存の分散環境への接続も頻繁に行なわれている。

しかしながら、電話回線や無線を利用したネットワークは、LAN などの既存のバックボーンネットワークと比較して、狭いバンド幅、高い通信コスト、頻発する通信切断などの様々な制約が存在する。現在どのような場所でも通信手段の確保が可能になりつつあるものの、これらすべての制約を満足させる移動体計算機環境に適した通信メディアは存在しない。そのため、ユーザが移動計算機環境上で、既存のバックボーンネットワーク上と同様にアプリケーションを実行することが不可能となっている。

このような通信上の制約に対処するため、近年ディスコネクティッドオペレーションに関する研究が行なわれている。移動計算機上のクライアントは、オフィスや学校にいる時は LAN などの専用線を利用してアプリケーションにアクセスしている。しかし外出先では、公衆の電話回線など制約の多い通信メディアを利用する場合が多い。そこで、外出する直前、すなわち LAN などの専用線との接続を断つ前に、移動計算機上にサーバ上のアプリケーションやファイルを複製としてコピーする。このように複製を保持することによって、オリジナルのサーバへのアクセスが困難あるいは不要な場合は、移動計算機上に複製したアプリケーションやファイルを利用することができる。そして通信回線が使用可能になった時、あるいは最新のデータが必要になった時に通信回線を接続して、オリジナルサーバへアクセスする。これにより、不要な通信回線の利用を回避することができ、全

体として高速でかつ低い通信コストでアプリケーションを利用することができる。こうした手段により、バンド幅が狭く、中断が頻発する通信回線上で、効率的にデータを送受信することが見かけ上、可能となっている。

従来の研究は性能、通信コスト、一貫性といった複数あるユーザやアプリケーションの要求の１つに焦点を当て、それぞれの最適化を試みているものが多い。その反面、現実のユーザやアプリケーションの要求は、すべてのメトリックスが満足されていることを理想としているため、これらの研究成果と現実の間の隔たりは甚だしいものとなっている。しかしながら、すべてのメトリックスを満足させることは、現実的には不可能である。このような研究成果と現実の間に見られる隔たりを小さくするためには、ユーザの複数ある要求のトレードオフを考慮することが不可欠である。

例えば、共有データの一貫性が重要であり、常に最新のデータにアクセスする必要があるならば、使用可能な通信メディアが存在する限り、通信コストが多少高くても回線接続を行なう必要がある。その一方で、あまり通信コストをかけることができず、最新のデータにアクセスする必要がないならば、一貫性を多少なりとも犠牲にして移動計算機上にある複製されたデータにアクセスすればよい。このようなことから、ユーザが用途に応じてトレードオフを考慮し、その結果を実装に反映させることができるならば、ユーザやアプリケーションの要求に適応したアプリケーションの実行が可能になると考えられる。

また、このようなトレードオフは将来にわたって存在し続けるものと考えられる。例えば、通信コストは年々下がりつつあるが、日進月歩の技術革新によってさらに高性能な通信メディアが登場し、それらは既存の通信メディアよりも相対的に通信コストが高くなっているというのが現状である。このような状況においてアプリケーションを快適に利用するために、ユーザは常に様々なメトリクッス間のトレードオフを考慮する必要性に迫られている。

本論文では、ユーザが各メトリクッス間のトレードオフを考慮・定義できる汎用的なインタフェースを提供するとともに、それらを複数の異なる実装へマッピングするためのフレームワークやメカニズムを提供する。

ユーザはこのインタフェースを用いて、考慮すべきメトリック種別とそのメトリックスの重要度を表す数値パラメタを指定する。インタフェースと実装をマッピングするために用意されたマッピングレイヤーが、指定されたパラメタを計算して実装種別を決定する。マッピングレイヤーと実装の間にもインタフェースが定義されており、マッピングレ

イヤーがこのインタフェースを通して、実装レイヤーがサポートしている実装種別を呼び出す。両インタフェースが定義されているため、これらのインタフェース仕様に適合する限り、マッピングレイヤーの内部を簡単に変更することができる。

これにより、ユーザやアプリケーションの要求に適応可能なアプリケーションを構築することが可能となる。

実際の検証作業としては、まずオブジェクト指向のアプリケーションに対してディスコネクティッドオペレーションの機能を提供するツールキットを用意した。このツールキットは、通信コストと共有データの一貫性のトレードオフに基づいた異なる実装をサポートしている。そして、アプリケーションの例として少人数で使用するグループスケジューラを作成し、実際の用途をシミュレーションすることにより、通信コストと共有データの一貫性のトレードオフをユーザが考慮することの有効性を実証した。

ツールキットの作成では、これまでほとんど重要視されてこなかった楽観的並行制御における競合解消について、様々な競合解消戦略を提案し、ユーザの要求に基づく一貫性維持の実現を目指した。さらに、従来の研究では全く指摘されてこなかったアプリケーションのセマンティクスへの依存性や、競合解消に伴なうシリアライザビリティの保証に関する問題点についても提起する。

第 2 章

移動計算機環境における適応可能なアプリケーションの構築

2.1 移動計算機環境の制約

現在、電子装置の小型化、高性能化、低消費電力化等の技術進歩により、PDA などの移動計算機上での様々なアプリケーションを利用するようになってきた。また、携帯電話や PHS 等の無線を利用した通信メディアの普及や、PIAFS 等に見られる通信回線の高速化により、移動先から会社や学校にある既存の LAN などの分散環境に接続して、データの送受信を行なうことも増えつつある。このように、既存の分散環境と融合して移動先でも利用可能な計算機環境は、移動計算機環境と呼ばれる。

しかしながら、移動計算機環境を利用しているユーザからは、通信が遅い、電話料金が高いなど様々な不満の声が挙がっているのもまた現状である。こうした不満や苦情は、移動計算機環境が従来の分散環境と異なる制約を有することに起因している。

ここでは、そうした移動計算機環境に特有な様々な制約について説明する。

2.1.1 ハードウェアに関する制約

移動計算機は持ち運んで使用するという用途から、小型化・軽量化が不可欠である。また、通常の計算機と同等の性能のハードウェアコンポーネントを提供しようとすると、それぞれの小型化・軽量化が求められ、コストが嵩む。そのため、コストとの兼ね合いか

ら、通常の計算機と比較してより低性能なものにならざるを得ない。以下にいくつかのコンポーネントにおける制約について述べる。

- メモリ

  移動計算機はメモリの小型化が必要なため、低コストで大容量のメモリを搭載することができない。また、メモリを搭載するスペースを十分に確保することができない。

- バッテリ

  通常の計算機ではＡＣ電源から電力を確保するため、電力不足の問題は起こらない。それに対して、移動計算機ではバッテリ駆動のため電力は無尽蔵ではなく、長時間使用しようとすると、節約が必要となる。この制約は、他のコンポーネントにも大きな影響を与える。

- ＣＰＵ

  高性能のＣＰＵはコストが高く、バッテリ消費量も大きくなるため、やや性能の劣るＣＰＵを搭載することになりがちである。

- ストレージ

  大量のディスクなどを搭載することができないため、大規模のアプリケーションを単独で実行することはできない。また、ハードディスクのようなストレージはバッテリ消費量が大きくなる。

- ディスプレイ

  小型で高解像度のディスプレイは、高価であり、かつバッテリ消費量が大きい。そのため、色数が減ったり、解像度のやや劣るディスプレイを使用するケースがある。

2.1.2 通信・ネットワークに関する制約

現在、どのような場所でも何らかの通信手段を確保することが可能になりつつあるが、移動先ではLAN のような通信メディアは通常使用できず、携帯電話や公衆電話回線などを利用する場合が殆んどである。そのため、以下のような制約が発生する。

- バンド幅

  通常の計算機環境で使用される通信メディアは、Ether では 10Mbps、FastEther や FDDI に至っては 100Mbps である。それに対して移動計算機環境で使用可能な通信メディアは、携帯電話が 9600bps、PHS の PIAFS が 32Kbps と、極端にバンド幅が小さい。

- 使用料金

  構内 LAN などでは通常、接続料金や回線料金はかからない。それに対して公衆の電話回線を使用した場合、接続回数や使用時間に応じて料金が加算される。

- 一時的切断

  移動計算機環境では携帯電話のような無線を利用することがある。無線を利用した通信メディアでは、一時的に接続がとぎれたりすることが頻発する。

- ハンドオフ

  通信メディアに無線を使用した場合、移動に伴なって移動計算機と直接通信していた基地局が切り替わることがある。この切り替えに対してアプリケーションを中断なく使用するためにはハンドオフ処理が必要となり、その遅延は大きくなる。

2.2 ディスコネクティッドオペレーション

前述の様々な制約、特に通信・ネットワークに関する問題に対処するために提案された機能の１つがディスコネクティッドオペレーション [1] である。

ディスコネクティッドオペレーションでは、予め必要を思われるサーバ上のアプリケーションやファイルを複製として自分の計算機上にコピーする。このように複製を保持することによって、オリジナルのサーバへのアクセスが困難あるいは不要な場合は、自分の計算機上に複製したアプリケーションやファイルを利用することができる。そして通信回線が使用可能になった時、あるいは必要になった時に通信回線を接続して、オリジナルサーバへアクセスする。サーバとの通信は、まとめて実行されるため、通信回線を使用する時間が大きく減少する。こうした手段により、バンド幅が狭く、中断が頻発する通信回線上で、効率的にデータを送受信することが可能となっている。

しかしながら、ここには考慮しなければならない問題がいくつか挙げられる。次にこれらの問題について検討する。

(1) プリフェッチ

必要と思われるファイルなどを自分の計算機上にコピーする際、どのファイルをコピーするかを予測することは非常に困難である。

そのため、優先度に基づいたキャッシュのアルゴリズムを利用したり [1]、アプリケーションがファイルをグループ化できる機構を提供する [9][10] など、様々な方法が提案されている。しかし、これらの方法は効率的ではあるが、必ず複製が存在することを保証しているわけではない。複製が存在しないということは、サーバに接続できない限り実行を中断せざるを得ないことを意味するため、アプリケーションを実行する上で大きな障害となる。

しかしながら最近は、障害によりサーバへ接続できない状態が長時間続くことは少なく、比較的必要な時に接続可能となっている。そのため、複製がない時は可能な限り即サーバへ接続できることを担保に、ＣＰＵやネットワークが空いている時にバックグランドで必要なファイルをプリフェッチするなどして、バンド幅が小さい通信メディアでの効率的なデータの送受信を実現している [6][7]。

(2) 一貫性維持

ディスコネクション中にクライアントが複製されたデータを更新した場合、サーバ上のオリジナルデータとの一貫性が保証されなくなる。そのためディスコネクティッドオペレーションをサポートするには、一貫性を維持する機構が必要となる。

一貫性を維持する方法として、悲観的並行制御と楽観的並行制御 [14] の２種類の制御方法があるが、ディスコネクティッドオペレーションでは通常楽観的並行制御を採用している。なぜなら、ロックを取得したまま回線を切断し、その時間が長くなると、他の接続している計算機がそのロックを取得できずに実行を中断されてしまうからだ [1]。このような理由により、ディスコネクション中は更新ログを残し、サーバ上に接続した時に一貫性を回復する楽観的並行制御が採用されている。

しかしながら、楽観的並行制御ではシリアライザビリティが保証されている訳ではなく、また更新ログを使用した一貫性の回復処理では競合するデータが存在する場

合がある。ファイルに関する競合では、比較的シンタクス依存のため競合を解消しやすい面がある。その反面、アプリケーションのセマンティクスへの依存性が高いオブジェクト指向のアプリケーションでは、競合解消においてそのようなセマンティクス情報が必要となる。そのため、何らかの精度の高い競合解消方法が必要となるが、競合が発生した時の対応はユーザに任せていることが多い [11][12]。

(3) トレードオフ

ディスコネクティッドオペレーションでは、必要なデータが複製データ中に存在しない場合、オリジナルのサーバへ接続している。そのことは裏を返せば、複製データの中に必要なデータが存在する限り、オリジナルサーバへは接続しないことを意味する。

しかしながら、ディスコネクションしている時間が長くなればなるほど、その間他のユーザがオリジナルのデータを変更する可能性が高くなる。そのため、ディスコネクションしているユーザは古くなったデータにアクセスする可能性がある。

使用しているデータが、最新でなければならないのか、それとも多少古くても構わないのかは、ユーザやアプリケーションの要求に依存するものであり、システム側で勝手に想定することはできない。

また、金銭に余裕のあるユーザにとっては、レスポンスが気にならない限り、ディスコネクションしている時間が短いほうが、高い一貫性が保証されるため都合がよい場合がある。その反面、金銭に余裕のないユーザにとっては、一貫性を犠牲にしてでも、ディスコネクション時間が長いほうが好ましい場合もある。

このようなトレードオフが現実に存在するため、システム側だけで最適なアプリケーションの実行を実現するのは不可能である。

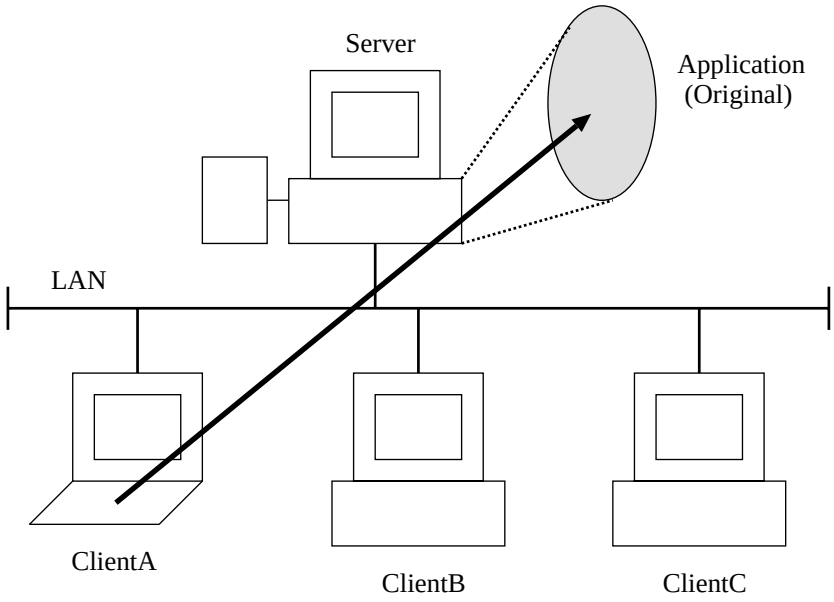


図 2.1: Connected Operation

2.3 適応可能なアプリケーションの構築

上述した既存のディスコネクティッドオペレーションの問題点を十分に検討した上で、本研究では、オブジェクト指向のアプリケーションに対してディスコネクティッドオペレーションの機能をサポートするツールキットの作成を試みた。

以下、上述の問題点に対応させながら、我々が試みているツールキットの特徴について述べる。

(1) プリフェッチ

現在、どのような場所でも何らかの通信手段を確保することが可能になりつつあり、かつ障害により接続できない状態が長時間続くことが比較的少なくなっている。さらに、サーバアプリケーションの複製がすべて移動計算機上にコピーできる規模のアプリケーションを想定することにより、プリフェッチの問題については追求しないこととした。

こうした想定のもとでは、以下のようにアプリケーションを使用することができる。

> 移動計算機上のクライアントは、オフィスや学校にいる時はLAN などの専用線を利用してアプリケーションにアクセスする（図 2.1）。外出など

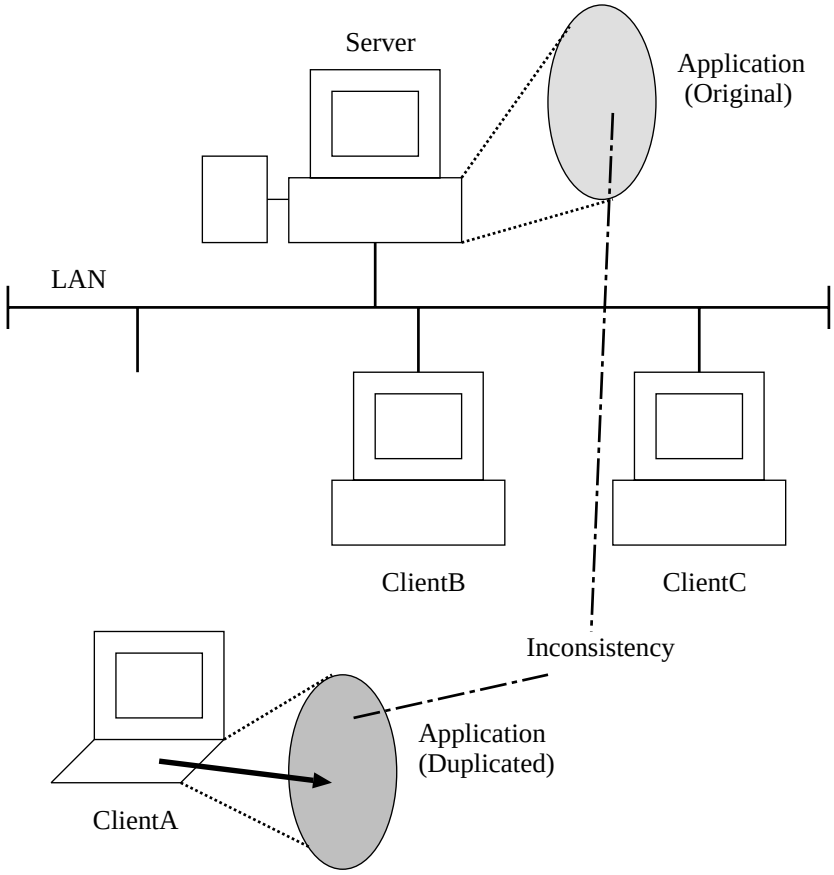


図 2.2: Disconnected Operation

により移動する際は、その直前、すなわち LAN などの専用線との接続を切断する前に、移動計算機上にサーバアプリケーションを複製としてコピーする。

移動先でオリジナルサーバへのアクセスが困難あるいは不要な場合は、移動計算機上に複製したサーバアプリケーションを利用する（図 2.2）。そして、最新のデータが必要になった時に通信メディアが使用可能ならば、通信回線に接続して、オリジナルのサーバへアクセスする。通信メディアが使用不可能ならば、とりあえず複製サーバを利用する。

一貫性の維持は、移動先でオリジナルサーバに接続した時、あるいは元の LAN などの環境に戻ってきた時に行なわれる。

(2) 一貫性維持

我々も既存の研究と同様、一貫性維持については楽観的並行制御を採用した。

一方、競合解消のためのいくつかの戦略を用意し、ユーザやアプリケーションがそれらの戦略を選択することににによって、彼らの要望に基づいた一貫性の維持を実現することを目指した。

さらに、ユーザ介入やコーディネーション機能の追加により、より柔軟性のある機構を用意した。

(3) トレードオフ

上述した通信コストと一貫性のようなトレードオフを、ユーザやアプリケーションが考慮できるようなインタフェースを用意した。

汎用インタフェースと、それを実装にマッピングできる機構を提案することにより、ユーザがシステマティクにトレードオフを記述し、それを実装に反映させることを目指した。

このようなツールキットをサポートすることにより、ユーザやアプリケーションの要求に適応可能なアプリケーションの構築が可能になると考えられる。

第 3 章

トレードオフを記述するインタフェース

3.1 トレードオフの考慮

あるインタフェースを提供する場合、1つの実装法がすべての要求を満足することは考えられない。常に、いくつかのメトリックスのトレードオフを考慮する必要がある。特に移動計算機環境のような様々な制約が課される環境においては、すべての要求を満足するインタフェースの実現は不可能である。ディスコネクティッドオペレーションをサポートする通信機構に対する要求を例にとってみても、通信コストと性能、通信コストと一貫性維持の重要度、性能と一貫性維持の重要度、通信コスト・性能とセキュリティの度合などといった様々なトレードオフが存在する。

多様な環境に適応するアプリケーションの構築を考える時は、異なるトレードオフを実現する複数の実装を環境が変化するごとに切替える必要がある。すなわち複数の実装を動的に切替えることにより, 適応可能なアプリケーションを構築することが可能となる。

そのためには、それぞれの実装が仮定するトレードオフを明らかにする必要がある。そうでないと、どの実装をいつ用いるかを決定することは不可能である。また、実装のトレードオフを定義するためのメトリックスを決定することも重要である.

そこでトレードオフをパラメタとして明確に定義・記述できるインタフェースを提供することにした。このインタフェースを使用することにより、実装側に優先されるべき要求が伝えられる。

3.1.1 通信コストと共有データの一貫性のトレードオフ

ここでは、ディスコネクティッドオペレーションにおける通信コストと一貫性維持の重要度のトレードオフを例にとり、具体的にトレードオフをどのように考慮するかについて説明する。

共有データの扱い

ディスコネクティッドオペレーションをサポートする移動計算機環境では、サーバ・クライアント間の共有データを如何に扱うかが重要な問題となる。並行制御についてはロックなどを使用する悲観的並行制御が一般的である。しかし移動計算機環境ではロックを保持したまま音信不通になることが起こるため、悲観的並行制御は適していない。そのため回線切断中は更新ログを残し、再接続時に一貫性の回復処理を行なうという楽観的並行制御が一般に行なわれている。

本研究でも更新ログを用いた楽観的並行制御を採用している。しかしながら、我々はよりアプリケーションの振舞に特化した実装を実現するため、一貫性維持に対する要求をトレードオフとしてインタフェースに記述できるようにした。

トレードオフの考慮

通信コストを最小限に押えたいという要求を優先すれば、回線切断の時間が長くなり、複製されたサーバアプリケーションを利用することが多くなる。その間複製された共有データを更新する頻度が高くなればなるほど、共有データの一貫性が失われ、再接続時に一貫性の回復処理が求められる。場合によっては、一貫性を完全に回復することが不可能となる。一方、一貫性維持に対する要求を優先するならば、通信メディアの障害が回避できる限り回線接続を維持し、オリジナルのサーバ上にある共有データを更新する。この場合、通信コストを押えたいという要求は満たされなくなる。

このように通信コストと一貫性維持の重要度の間には、ユーザやアプリケーションからの指定なくしては解消できないトレードオフが存在する。なぜなら最適な実装はユーザの要求やアプリケーションの振舞に依存するからである。

そこでユーザやアプリケーションがトレードオフを指定・記述することにより、その要求を実装側に伝える。実装側はこの要求に基づき、いつオリジナルのサーバ側に再接続して一貫性回復の処理を行なうかを決定する。これにより、ユーザの要求やアプリケーショ

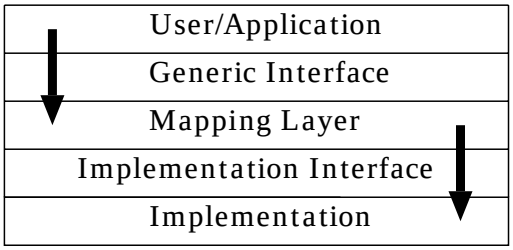


図 3.1: トレードオフを記述するためのフレームワーク

ンの振舞に適応可能な実装を実現することが可能となる。

3.2 フレームワーク

トレードオフを記述するインタフェースの仕様について詳述する前に、まずそのためのフレームワークについて説明する。

図 3.1は、アプリケーションやシステムをサポートする実装、トレードオフ記述のインタフェースなどの関係を示している。このフレームワークでは、インタフェースを実装にマッピングするレイヤーを、マッピングレイヤーとしてアプリケーションと実装の間に定義している。

このマッピングレイヤーは２種類のインタフェース・レイヤーを通じて、最適な実装を選択するためのパラメタをやりとりする。１つのインタフェースは、ユーザやアプリケーションがトレードオフを記述するための汎用インタフェースであり、ユーザやアプリケーションの要求をマッピングレイヤーに伝える役割を果たす。もう１つのインタフェースは、マッピングレイヤーが決定した実装を、システムをサポートする実装自身に伝えるための内部インタフェースである。

以上の２つのインタフェースを通して、トレードオフに関するユーザやアプリケーションの要求を実装側に伝えることが、このフレームワークの骨子となっている。

以下この章では、インタフェースの記述仕様を示すとともに、各レイヤーの詳細について説明する。

3.3 汎用インタフェース

3.3.1 メトリックスの重要度の設定

汎用インタフェースはトレードオフの種類やサポートする実装の種類に依存しない記述仕様である。各メトリックに対してパラメタ値を設定することにより、ユーザやアプリケーションの要求を実装側に伝えることができる。

以下にその仕様を示す。

SetTradeoff(メトリック種別, パラメタ値)

機能	:	指定されたメトリックにパラメタ値を設定する。
メトリック種別	:	トレードオフの種類にかかわりなくメトリックを １つ指定する。
パラメタ値	:	指定されたメトリックの重要度を表す数値を 0.00 から 1.00 の範囲で指定する。 ($0.00 \leq$ パラメタ値 ≤ 1.00)

あるトレードオフは複数のメトリックスのそれぞれの重要度によって表わされるため、実際に使用する時はこの手続きを複数呼び出す必要がある。例として、通信コストと一貫性のトレードオフについてその重要度を指定する場合は、

SetTradeoff(通信コスト, 0.80)

SetTradeoff(一貫性, 0.10)

のように２つのメトリックスの重要度を記述する。

この時、実際に最適化される実装はサポートするシステムの実装の種類に依存するものであり、我々がサポートするディスコネクティッドオペレーションでは接続の可否や接続時期を決定するために利用される。そのような依存性を透過的にする手段として、インタフェースと実装間でのマッピングを行なうレイヤーが必要となるが、これについては後述する。

このようなインタフェースを提供することにより、ユーザはシステマティックに各メトリックの重要度を指定することが可能となる。

3.3.2 その他の記述仕様

実際にユーザに提供されるインタフェースとしては、各メトリックにパラメタ値を設定する *SetTradeoff* の他に以下のような手続きがある。

- *GetTradeoff(*メトリック種別*)*

  指定されたメトリックの現在のパラメタ値を参照する。戻り値としてパラメタ値が返される。

- *ClearTradeoff(*メトリック種別*)*

  指定されたメトリックの現在のパラメタ値を無効とし、デフォルトの値を設定する。

- オブジェクト単位での操作

  より綿密で最適な実行を可能にするため、アプリケーション単位だけではなく、オブジェクト単位で各メトリックのパラメタ値を設定できる。

  - *SetTradeOffObject(*メトリック種別*,* オブジェクト*,* パラメタ値*)*

    指定された１つのオブジェクトに対し、メトリックのパラメタ値を設定する。

    対応する手続きとして、現在のパラメタ値を参照する *GetTradeOffObject (*メトリック種別*,* オブジェクト*)* や、現在のパラメタ値を無効にしてデフォルト値を設定する *ClearTradeOffObject(*メトリック種別*,* オブジェクト*)* がある。

  - *SetTradeOffObjectS(*メトリック種別*,* オブジェクト数 *,* オブジェクト *1 ,* オブジェクト *2 , ... ,* パラメタ値*)*

    指定された複数のオブジェクトに対し、メトリックのパラメタ値を設定する。

    対応する手続きとして、現在のパラメタ値を参照する *GetTradeOffObjectS (*メトリック種別*,* オブジェクト数 *,* オブジェクト *1 ,* オブジェクト *2 , ...)* や、現在のパラメタ値を無効にしてデフォルト値を設定する *ClearTradeOffObjectS(*メトリック種別*,* オブジェクト数 *,* オブジェクト *1 ,* オブジェクト *2 , ...)* がある。

  - *SetTradeOffGroup(*メトリック種別*,* グループ*,* パラメタ値*)*

    指定されたオブジェクトのグループに対し、メトリックのパラメタ値を設定する。

対応する手続きとして、現在のパラメタ値を参照する *GetTradeOffGroup (メトリック種別, グループ)* や、現在のパラメタ値を無効にしてデフォルト値を設定する *ClearTradeOffGroup(メトリック種別, グループ)* がある。

3.4 特殊インタフェース

特殊インタフェースはトレードオフの種類やサポートする実装の種類に依存する記述仕様であり、前記の汎用インタフェースを補強することを目的としている。

以下にその仕様を示す。

SetTradeoffSpecific(メトリック種別 1, メトリック種別 2, 識別子, [値 1, 値 2, ...])

機能 : メトリック種別 1 とメトリック種別 2 の間のトレードオフについて指定された識別子に関わる実装上の要因を考慮する。

値 : 識別子に応じて必要とされる値を指定する。

使用例として、通信コストと一貫性のトレードオフに関して、10 時には必ず一貫性を維持しているようにしたい場合は、

SetTradeoffSpecific(通信コスト, 一貫性, 接続時期指定, 10:00)

のように記述する。

また、食事中は接続したくない場合、

SetTradeoffSpecific(通信コスト, 一貫性, スケジュール指定, 接続拒否、食事)

のように記述し、スケジューラと連動して食事時間の接続を避けることも可能となる。

その他に提供される特殊インタフェースとしては、現在特殊インタフェースを使用して設定されている状況を参照するための *GetTradeoffSpecific* や、設定を無効にする *ClearTradeoffSpecific* などがある。

汎用インタフェースと特殊インタフェースの設定の優先順については、特殊インタフェースにおける設定がまず優先される。特殊インタフェースで設定された条件が満たされない場合、汎用インタフェースを用いて設定されたパラメタ値を参照し、実装を最適化する。

3.5 マッピングレイヤー

トレードオフの種類やシステムをサポートする実装の種類に依存しないインタフェースを提供する場合、インタフェースの記述を実装にマッピングするための機構が必要となる。このマッピング機構を提供しているのがマッピングレイヤーであり、このレイヤーを挿入することにより、ユーザやアプリケーションに対して透過的なインタフェースの提供が可能となる。

その一方、実装をサポートする側は適応可能な実装を実現するために、マッピングレイヤーを定義・実装する必要がある。そこで、ここでは一般的なマッピングの方法を示すとともに、実際にその方法をどのように適用していくかについて述べる。

3.5.1 マッピングの方法

メトリックAとメトリックBの間のトレードオフをマッピングする場合、まず汎用インタフェースを通して得られたメトリックAとメトリックBのパラメタ値M_A、及びM_Bから、その差分のパラメタ値P_{AB}を以下のように求める。

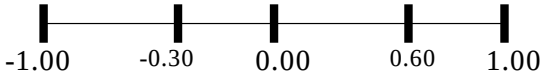
$$P_{AB} = M_A - M_B$$
$$-1.00 \leq P_{AB} \leq 1.00$$
$$P_{AB} : AB\text{間のトレードオフをマッピングするためのパラメタ値}$$

ここで算出されたパラメタ値P_{AB}は、それが1.00に近ければ近いほどAを重要視し、-1.00に近ければ近いほどBを重要視していることを意味する。

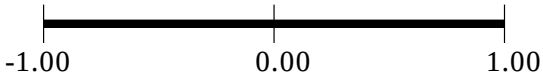
次に、このパラメタ値P_{AB}を以下に示す３つの形態のどれかを利用して実装I_{AB}へのマッピングを定義する。

(1) 離散的なマッピング

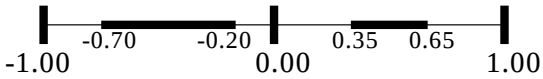
実行されるモジュールが違うなど、明らかに実装の質が異なる場合に採用されるマッピング方法。図3.2の例では、５つの質の異なる実装がマッピングされている。



(1). Discrete Mapping



(2). Continuous Mapping



(3). Combined Mapping

図 3.2: マッピングの種類

$$I_{AB} = \begin{cases} I_{AB_{11}} & : & -1.00 \\ I_{AB_{12}} & : & -0.30 \\ I_{AB_{13}} & : & 0.00 \\ I_{AB_{14}} & : & 0.60 \\ I_{AB_{15}} & : & 1.00 \end{cases}$$

I_{AB} : AB間のトレードオフに対応する実装

(2) 連続的なマッピング

あるパラメタの値を変更するのみなど、量的な度合の相違から実装が連続的に変化する場合に採用されるマッピング方法。図 3.2の例では、質的には等しい１つの実装が、量的な相違でもって連続的にマッピングされている。

$$I_{AB} = I_{AB_{21}} : [-1.00, \quad 1.00]$$

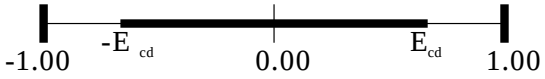


図 3.3: マッピングの具体例

(3) 離散・連続の融合マッピング

上記の２つの形態を融合したマッピング方法。これにより、さらに柔軟性に富んだ実装へのマッピングが可能となる。図 3.2の例では、５つの質の異なる実装がマッピングされ、そのうちの２つの実装 ($I_{AB_{32}}, I_{AB_{34}}$) は量的な相違が考慮されている。

$$I_{AB} = \begin{cases} I_{AB_{31}} & : & -1.00 \\ I_{AB_{32}} & : & [-0.70, \quad -0.20] \\ I_{AB_{33}} & : & 0.00 \\ I_{AB_{34}} & : & [0.35, \quad 0.65] \\ I_{AB_{35}} & : & 1.00 \end{cases}$$

3.5.2 マッピングの具体例

ここでは、上述されたマッピングの方法が実際にどのように適用されるかについて、通信コストと一貫性のトレードオフを例に取り挙げ詳述する。

この具体例では、離散・連続の融合マッピングの方法を用いて実装へのマッピングを行なっている。実装形態としては、質の異なる３つの実装が提供されており、うち１つは量的な相違が考慮された実装になっている。

$$I_{CD} = \begin{cases} I_{CD_1} & : & -1.00 \\ I_{CD_2} & : & [-E_{CD}, \quad E_{CD}] \quad , \quad 0.00 < E_{CD} < 1.00 \\ I_{CD_3} & : & 1.00 \end{cases}$$

I_{CD} : 通信コスト (C) と一貫性 (D) の間のトレードオフに対応する実装

以下にそれぞれの実装に関する説明を述べる。

(1) I_{CD_1} : 一貫性を最大限に重視する実装（通信コスト無視）

通信コストを考慮せず、可能な限り回線接続を維持する。

(2) I_{CD_2}：両メトリックスのパラメタ値に基づく実装

両メトリックスのパラメタ値に基づいて接続間隔時間を決定し、それに従い定期的に回線接続を行ない、一貫性の維持を行なう。接続間隔時間の決定方法については後述する。

(3) I_{CD_3}：通信コストを最大限に重視する実装（一貫性無視）

ディスコネクションする以前と同じ通信メディアが使用可能になるまで、すなわち移動前の環境に戻るまで回線接続を行なわない。

次に連続的なマッピングである実装 I_{CD_2} に関して、実際どのようにして接続間隔時間を決定するかについて説明する。

接続間隔時間は以下の式を用いて算出される。

$$Interval(P_{CD}) = \begin{cases} \dfrac{P_{CD}}{E_{CD}} \times (Max - Standard) + Max & : \quad P_{CD} \geq 0 \\ \dfrac{P_{CD} - E_{CD}}{-E_{CD}} \times (Standard - Min) + Min & : \quad P_{CD} \leq 0 \end{cases}$$

$Interval$ ：接続間隔時間（分）
$Standard$ ：標準接続間隔時間（分），　$P_{CD} = 0.00$ の時の接続間隔時間
Max ：最大接続間隔時間（分），　$P_{CD} = E_{AB}$ の時の接続間隔時間
Min ：最小接続間隔時間（分），　$P_{CD} = -E_{AB}$ の時の接続間隔時間

この式をグラフで表したものが、図 3.4である。

ここで実際に数値をあてはめて考えてみると、

$E_{CD} = 0.80, Standard = 60, Max = 240, Min = 15$ とすると、

$$Intarval(-0.80) = 15 \text{ 分}$$

$$Intarval(-0.40) = 37.5 \text{ 分}$$

$$Intarval(0.00) = 60 \text{ 分}$$

$$Intarval(0.40) = 150 \text{ 分}$$

$$Intarval(0.80) = 240 \text{ 分}$$

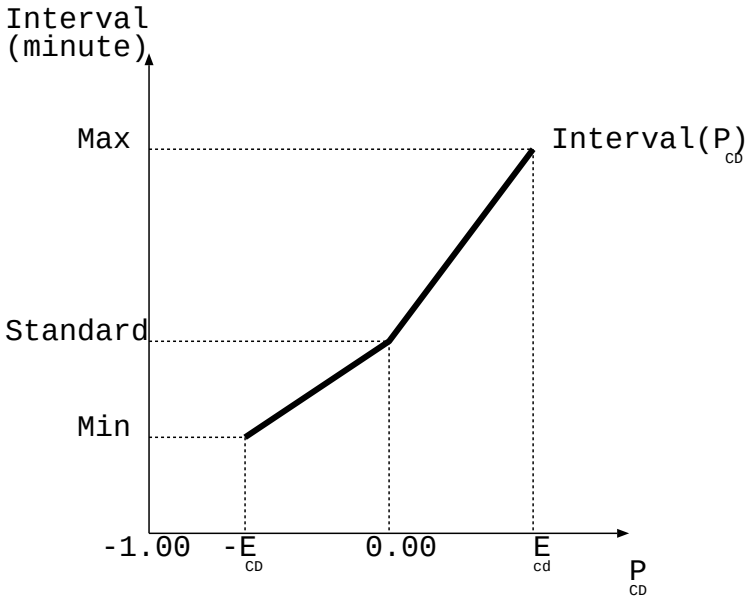


図 3.4: 接続間隔時間決定関数

となり、両メトリックスのパラメタ値に応じた接続間隔時間を求めることができる。

接続間隔時間決定関数は、使用するアプリケーションに応じて変更することも可能である。また、上記の関数を用い、標準・最大・最小接続間隔時間を変更するような簡単なチューニングも可能である。

3.6 マッピングレイヤーと実装間のインタフェース

これまで、ユーザやアプリケーションがトレードオフを記述するインタフェースの仕様や、それを実装側に伝えるためのマッピングレイヤーの構造について詳しく述べてきた。しかしながら、マッピングレイヤーで決定された実装種別をどのようにして実装側に伝えるかについては何も言及していない。

そこで、ここではマッピングレイヤーと実装の間のインタフェースについて説明する。

3.6.1 インタフェース記述仕様

ここで定義するインタフェースは、トレードオフの種類やシステムをサポートする実装の種類に依存しない汎用的なインタフェースである。

以下にその仕様示す。

SetImplementation(実装種別 [, パラメタ値])

機能	:	指定された実装を選択する。
実装種別	:	実装側が提供する質的相違のある実装の種別を指定する。
パラメタ値	:	実装種別が連続的なマッピングを許容している場合、 その量的相違を示す数値を指定する。

前出したマッピングの具体例を用いて説明すると、離散的なマッピングが行なわれる実装 I_{CD_1} を選択する場合、

SetImplementation(I_{CD_1})

のように記述する。

また、連続的なマッピングが行なわれる実装 I_{CD_2} を選択し、接続間隔時間 $Interval(P_{CD})$ が 60 分の場合、

SetImplementation(I_{CD_2}, **60**)

のように記述する。

その他に提供されるインタフェースとしては、現在このインタフェースを使用して実行されている実装種別を参照するための *GetImplementation* や、現在の実装を無効にして、デフォルトの実装を選択する *ClearImplementation* などがある。

また、オブジェクト毎の設定も用意されている。

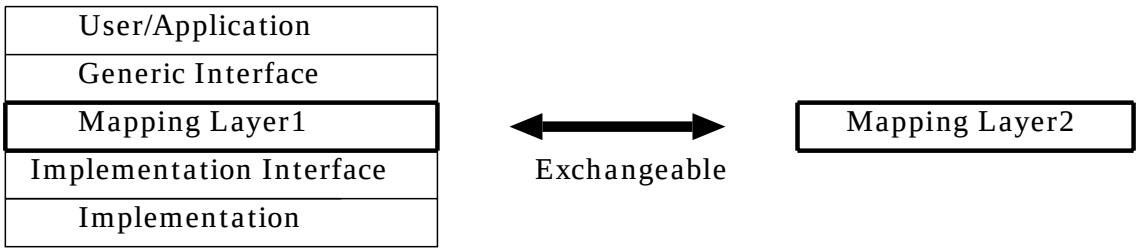


図 3.5: 交換可能なマッピングレイヤー

3.7 フレームワーク導入の利点

これまで述べてきたフレームワークでは、マッピングレイヤーの導入とともに、マッピングレイヤーの上位層と下位層のインタフェースを定義した。これにより、マッピングレイヤーの上位層にあたるアプリケーションや下位層にあたるシステムの実装を変更することなく、マッピングレイヤーの内部モジュールを変更したり、レイヤー自身をすべて置き換えることが可能となる。その様子を示したのが図 3.5である。

この透過的な交換可能性により、メモリやネットワークなどの計算機環境に変化があった場合、ユーザの負担を強いることなく、マッピングレイヤーを交換するだけで最適な実装を選択することができる。これにより、統合化された、さらに適応可能なアプリケーションの構築が実現可能となる。

第 4 章

設計と実装

4.1 ツールキットの構成

4.1.1 ツールキットの概要

まずツールキット全体の概要について、図 4.1に基づいて説明する。

既存のバックボーンネットワークから離れる直前に、ユーザはサーバアプリケーションをサーバ側から自分の移動計算機上にコピーする。回線切断中は、この複製されたサーバアプリケーションにアクセスすることにより、不必要な通信回線接続を回避することができる。ユーザは必要に応じて提供されたライブラリを通して、通信コストと一貫性のトレードオフを記述する。一貫性維持を行なうサーバ側への接続時期は、このユーザによるトレードオフの記述によって決定される。このユーザの要求に基づいてどちらのデータをアクセスすべきか判断しているのが、ディスコネクティッドオペレーションマネージャ（以下、ＤＯマネージャ）である。

クライアントからサーバアプリケーション上のオブジェクトへの更新要求があった場合、その要求はトレードオフを記述したインタフェースライブラリを通してＤＯマネージャに送られる (1)。ＤＯマネージャは、ユーザが設定したパラメタ値が格納されているアプリケーション要求テーブルから適切な回線接続時期を決定する。即座に接続する必要がない場合は、ユーザの移動計算機上にある複製サーバアプリケーション上のオブジェクトを更新し (2)、データの一貫性を維持するためのログに当該オブジェクトの更新を記録する。

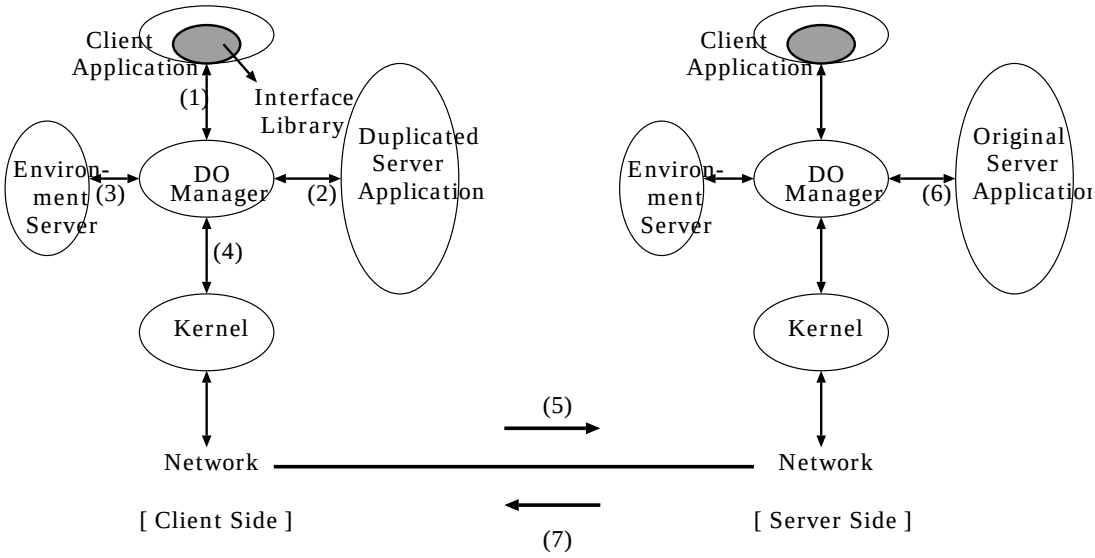


図 4.1: ツールキットの構成

一方即座に接続する必要がある場合は、現在接続可能な通信メディアが存在するかどうかを環境サーバ [18] に問い合わせる (3)。環境サーバは通信メディアの接続・切断やコストなどの計算機に関する実行環境情報を動的に保持しており、他のタスクやアプリケーションにそれらの情報を提供している。接続可能な通信メディアが存在する場合、サーバ側へ接続を試み (4)、ディスコネクション中に生成した更新ログをサーバ側へ送信する (5)。サーバ側のＤＯマネージャでは、更新ログを用いて競合の検出を行ない、競合解消結果に基づいて、オリジナルのサーバアプリケーションを更新する (6)。また、その結果はログの形でクライアント側にも戻され (7)、複製サーバアプリケーションも更新される。これにより、この時点での一貫性の維持が保証される。

しかしながら通信障害等によりサーバへの即時接続が不可能な場合、ＤＯマネージャは複製サーバアプリケーション上のオブジェクトを更新し、ログに記録することになる。この場合、通信メディアが使用可能になった時点でＤＯマネージャに通知するよう環境サーバに登録するとともに、その旨をユーザにも通知する。これにより通信障害等が起こった場合も、ユーザは中断なくアプリケーションを使い続けることが可能となる。

なお、このツールキットは RT-Mach[17] 上でＣ言語により実装されている。

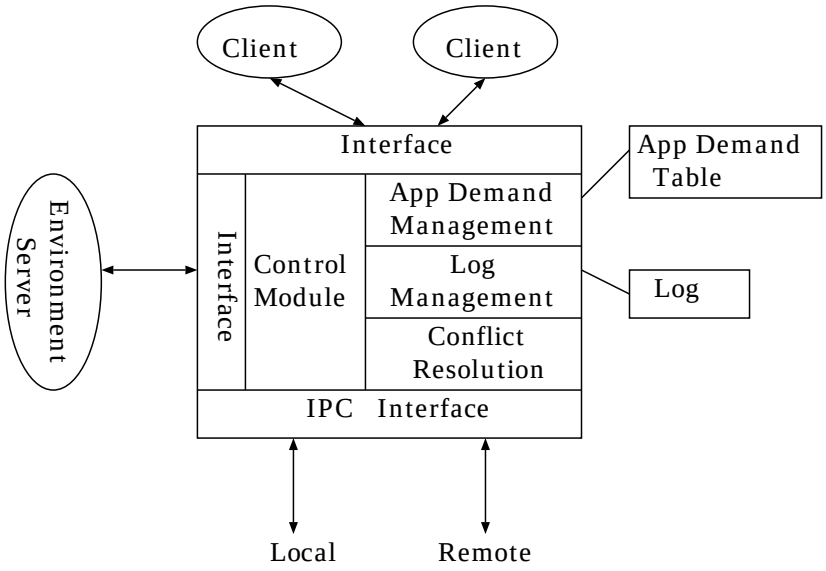


図 4.2: ＤＯマネージャの構成

4.1.2 ＤＯマネージャ

図 4.2に示されるＤＯマネージャは、ディスコネクティッドオペレーションをサポートするツールキットの中核をなすとともに、トレードオフの記述を実装にマッピングするマッピングレイヤの役割も果たしている。

このＤＯマネージャは主に以下の機能から成り立っている。

(1) アプリケーション要求管理

トレードオフを記述するインタフェースを通して得られたユーザやアプリケーションからの要求を受け付け、サーバ側への接続時期という実装を決定するマッピングレイヤのモジュール。

アプリケーション全体に対してだけではなく、各オブジェクトに対してもトレードオフを記述できるため、各オブジェクトごとに設定されたパラメタ値などが、アプリケーション要求テーブルを用いてここで管理されている。

(2) ログ管理

ディスコネクション中に発生した更新操作に対するログを管理するモジュール。

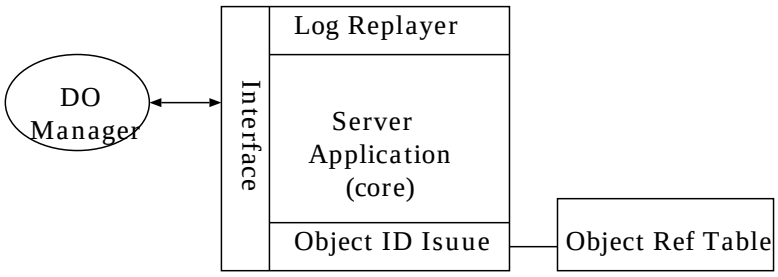


図 4.3: サーバアプリケーションの構成

サーバアプリケーションへのアクセスは必ずＤＯマネージャを介して行なわれるため、現状ではログの生成もこのモジュールで行なわれている。

(3) 競合解消

ディスコネクション中に生成されたクライアント側とサーバ側の更新ログを比較し、競合を検出・解消する一貫性維持モジュール。

サーバ側のＤＯマネージャにのみ存在し、クライアント側のＤＯマネージャには存在しない。

ユーザやアプリケーションが競合解消戦略を指定できるようにするため、ここで戦略の管理や実行を制御する。競合解消のためのコーディネーション機能もここに含まれる。

(4) IPC インタフェース

環境サーバから得られる回線の接続･切断情報や、アプリケーション要求管理モジュールの要求に従い、リモート上にあるサーバ側のＤＯマネージャやローカル上にある複製サーバアプリケーションへの接続を切り替えるモジュール。

サーバ側にあるＤＯマネージャでは、ディスコネクション中のクライアント側に接続を要求したり、オリジナルのサーバアプリケーションに接続したりする。

4.1.3 サーバアプリケーションの構成

図 4.3に示されるのがサーバアプリケーションの構成である。ディスコネクティッドオペレーションの機能をサポートするために、サーバアプリケーション本体の他に以下のよ

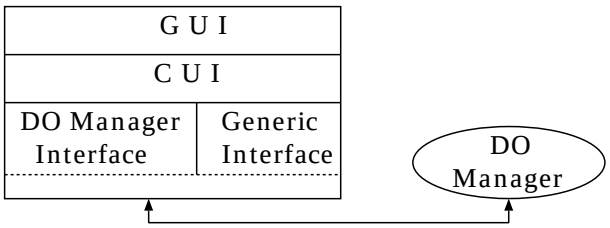


図 4.4: クライアントアプリケーションの構成

うなモジュールが組み込まれている。

(1) オブジェクトＩＤ発行モジュール

オリジナルサーバと複製サーバの間でオブジェクトを一意に識別するためのオブジェクトＩＤと、オブジェクト実体のアドレスをマッピングするモジュール。

新規オブジェクトの生成時にオブジェクトリファレンステーブル内にマッピングした結果を保存する。必要に応じてこのテーブルを参照することにより、オブジェクトＩＤやオブジェクト実体のアドレスを取得することができる。

オリジナルサーバで発行されるオブジェクトＩＤが正式なものであり、ディスコネクション中に複製サーバで発行されるものは暫定的なＩＤである。

(2) ログ・リプレイヤー

一貫性維持のプロセスにおいて、競合解消の結果のリプレイログをサーバアプリケーションに反映させるモジュール。

複製サーバ側では、回線接続が一時的なものである場合、オリジナルサーバのリプレイに続いて、サーバ側のＤＯマネージャから送られてきたリプレイログを用いて複製サーバをオリジナルサーバと同じものに更新する。リプレイログはオリジナルサーバ用と複製サーバ用とでは通常異なるものとなる。

4.1.4 クライアントアプリケーションの構成

図 4.4に示されるのがクライアントアプリケーションの構成であり、以下のようなモジュールが主に組み込まれている。

(1) ＣＵＩ・ＧＵＩ

クライアントアプリケーションの本体。

入力値チェックやオブジェクトＩＤと表示アイテムのマッピング機能などもここに含まれる。

(2) 汎用インタフェースライブラリ

トレードオフを記述する汎用インタフェースを使用するためのライブラリ。

(3) ＤＯマネージャインタフェースライブラリ

ＤＯマネージャを介してサーバアプリケーションにアクセスするためのライブラリ。

ディスコネクティッドオペレーションの機能を使用する場合、サーバ呼び出しに代わってこのインタフェースライブラリを呼び出す。

4.2　一貫性維持

ここではディスコネクティッドオペレーションをサポートする上で必要不可欠な一貫性維持機構について述べる。

更新ログを使用した楽観的並行制御による一貫性維持では、回線切断中の共有データの更新をログとして記録し、再接続時に一貫性の回復を行なう。そのため、サーバとクライアント側の両者で同一の共有データを更新した場合、一貫性回復時に競合が発生する。両者の間で競合が存在しない場合、クライアント側での属性値変更やオブジェクト生成に関する操作がサーバ側に反映される。なお接続が一時的であり、あるいはバックボーンネットワークに直結しない場合は、一貫性回復後直ちに回線を切断する。そのためサーバ側の変更をクライアント側にも反映させる必要がある。バックボーンネットワークに復帰する場合はその必要がなく、クライアント側にコピーされていたアプリケーションは破棄される。

両者の間で競合が存在する場合、競合の検出と解消のプロセスが必要となる。ファイルと違い、オブジェクトにおける競合検出では、アプリケーションのセマンティクスに依存した明確な競合の定義が必要である。その定義なくしては競合解消の結果がユーザの意図と相容れないものとなってしまう。同様の問題は、競合解消結果をサーバに反映させる段階においても発生し、単純に対応するオブジェクト内の指定位置のデータを変更するだけでは不十分となってしまう場合がある。

競合解消の手続きに関して従来の研究では、競合結果をユーザに示し、ユーザの手による競合解消を行なっていた。ここでは、ユーザ任せの競合解消方法だけでなく、ユーザの優先度や更新時間に基づく競合解消戦略を導入し、競合解消の部分的自動化機構を提供することにした。その他、コーディネータモジュールを導入することにより、競合解消に関するユーザ間の交渉を仲介したり、オリジナルサーバ内のオブジェクトの変更を他のユーザに通知する機能もサポートすることにした。

以下では一貫性維持の各処理についてその詳細を述べる。

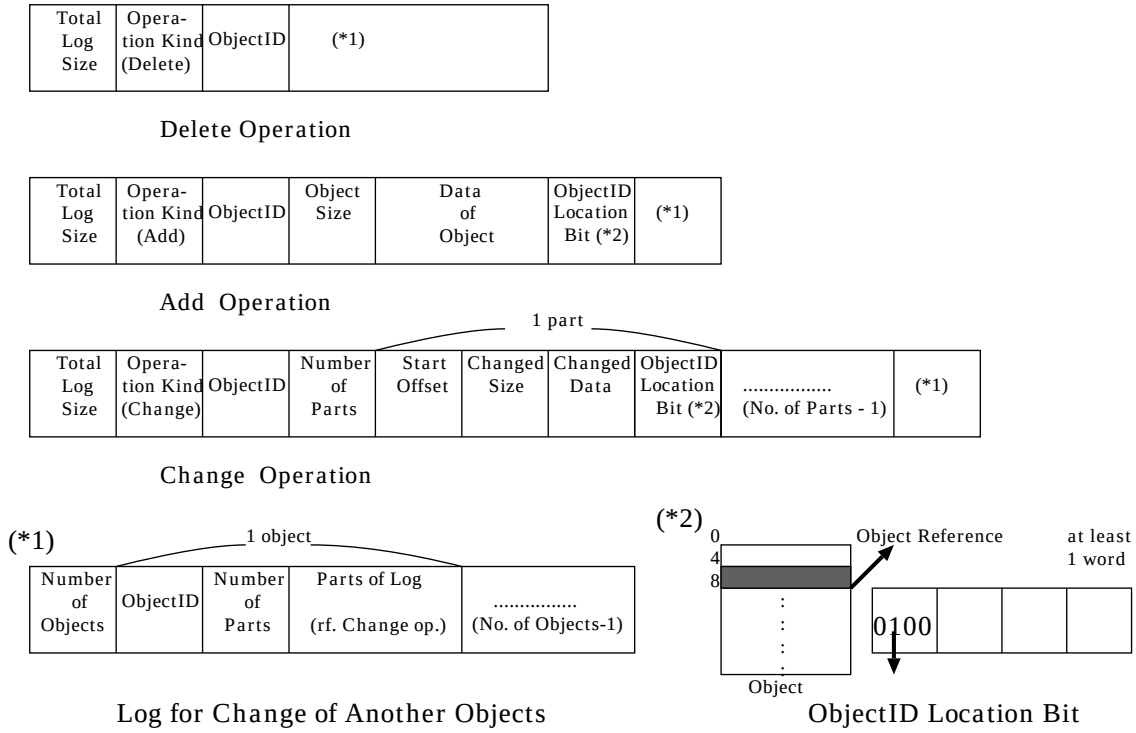


図 4.5: 各操作の更新ログ

4.2.1 更新ログの作成

ディスコネクション中のオブジェクトの更新に関する操作をログとして記録する。
図 4.5に削除系・追加系・変更系の３種類の操作に対応するログの内容を示す。
なお、(*1) と (*2) に対する補足説明は以下の通りである。

(*1) 操作に伴う他オブジェクトの変更ログ

このログは単独で現れることはなく、必ずある操作のログに付随するものである。例えばあるオブジェクトを削除した場合、そのオブジェクトを参照していたオブジェクトが存在すれば、通常そのポインター・データが変更される。このログは、こうしたオブジェクトの操作に連動した他のオブジェクトの変更を記録するためのものである。

(*2) オブジェクトＩＤ位置ビット

変更あるいは追加されたデータがオブジェクトのリファレンス（アドレス）である

場合、そのリファレンス値はログを生成した側でのみ有効である。そのため、リファレンス値をオブジェクトＩＤに変換する必要がある。しかしながら、オブジェクトＩＤは単なる整数型のデータであるため、ログをリプレイする側では、それがオブジェクトＩＤであることを認識できない。そのため、ログの中にこのオブジェクトＩＤ位置ビットを用意し、ログをリプレイする側がその整数値がオブジェクトＩＤであることを認識できるようにする。

図 4.5の例では、オブジェクトの４バイト目がオブジェクトのリファレンスであることを示している。ここではリファレンスのサイズが４バイトであるため、左から２つ目のビットがオンになっている。

オリジナルのサーバ側でも、ディスコネクティッドオペレーションを実行しているユーザがいる間は更新ログを生成するとともに、更新前のデータをトランザクションログとして記録する。

なお、オブジェクト変更のタイミングをどのように捉えるかについては、今のところ十分な検討がなされていない。現状ではアプリケーションに依存した形で対応している。

4.2.2 更新ログのマージ

ディスコネクション中に生成した更新ログは、マージを実行することにより、最適化することができる。現状ではタイムスタンプに基づく競合解消戦略をサポートしておらず、また楽観的並行制御におけるシリアライザビリティを考慮していない。そのため、同じオブジェクトＩＤのログが複数存在する場合は、１つのログにマージすることによって最適化を行なっている、

以下に現状で行なっているマージの規則を示す。

- 追加系のログの後に削除系のログが追加された場合、両者のログが削除される。
- 追加系のログの後に変更系のログが追加された場合、追加系のログに上書きされ、変更系のログは削除される。
- 変更系のログの後に変更系のログが追加された場合、１つの変更系のログに併合される。
- 変更系のログの後に削除系のログが追加された場合、変更系のログが削除される。

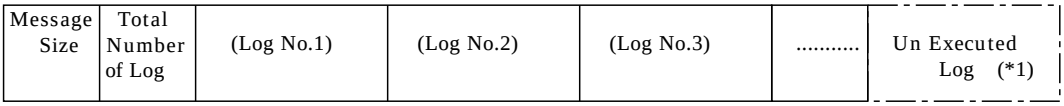


図 4.6: 更新ログ・ストリームのプロトコル

4.2.3 更新ログの送信

オリジナルのサーバへ接続する場合、図 4.6に示されるプロトコルを使用して、クライアント側のＤＯマネージャからサーバ側のＤＯマネージャへ更新ログがまとめて送信される。

(Log No.1)、(Log No.2) は更新ログの生成時に書き出される１つ１つの更新ログを示している。また、(*1) は未処理の更新ログであり、変更系操作が行なわれるオブジェクトがサーバ側への即接続を要求している場合に使用される。この時、複製サーバの更新は行なわれていないが、１つのログとしてサーバ側へ送信される。

なお、サーバ側から送り戻される競合解消結果のリプレイログにもこのプロトコルが使用される。また、そのログをＤＯマネージャからサーバアプリケーションに送信する時もこのプロトコルが使用される。

4.2.4 競合検出と解消

競合の検出と解消に関する手続きは、回線を切断する前から存在していた既存のオブジェクトに対するフェーズ、ディスコネクション中に新規に生成・追加されたオブジェクトに対するフェーズ、そして既存のオブジェクトと新規のオブジェクト間に対するフェーズから成り立っている。

以下にこれら３つのフェーズについて詳述する。

(1) 既存のオブジェクトに対するフェーズ

ここでは変更・削除系のログの中から、同一オブジェクトＩＤをもつログ同士の比較を行なうことによって競合を検出し、解消する。

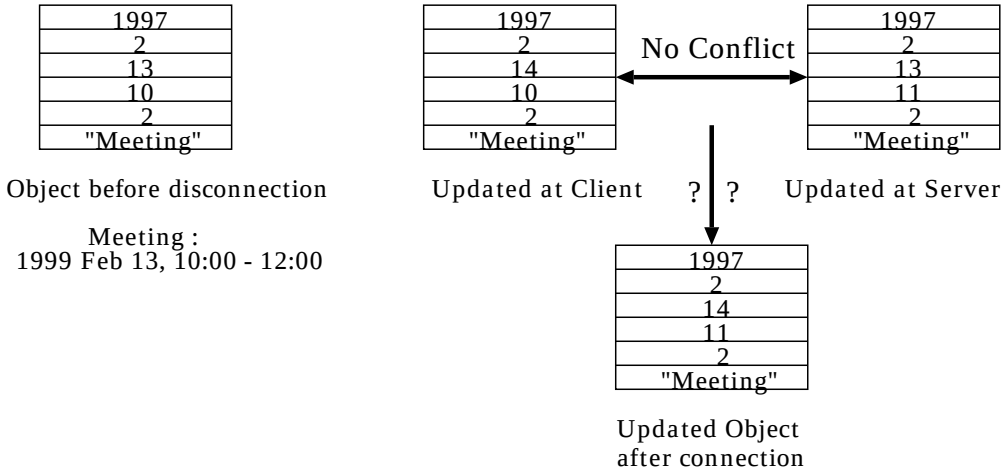


図 4.7: 同一オブジェクト内の競合

まず、ここで考慮しなければならないことは、同一オブジェクトに対する異なる位置の変更をどう扱うかということである。

ファイルのようにアプリケーションのセマンティクスに依存しないデータならば、競合とは認識せず両者の変更をマージしてしまうのが最も一般的である。オブジェクトベースのアプリケーションにおいても、このように１つのオブジェクト内にある複数のデータ間に依存関係が存在しない場合は、競合なしと判断することが可能である、しかしながら、依存関係が存在する場合は、競合状態となる。

依存関係が存在する例を、スケジューラを用いて説明すると図 4.7のようになる。

クライアント側ではディスコネクション中、会議の予定を１日繰り上げる変更をしたのに対し、サーバ側では１時間繰り上げる変更を行なった。それぞれ同一オブジェクトの異なる位置のデータを変更している。この際競合なしと判断すると、正式に更新されたスケジュールでは、この会議が１日と１時間繰り上げられたことになり、全く意味のないスケジュールとなってしまう。無意味になってしまう理由は、日と時の２つの属性間に依存関係が存在するからである。ここでは両者の変更を競合と判断するのが適当であり、どちらかの変更のみを採用するのが妥当である。

このようにオブジェクトベースのアプリケーションでは、オブジェクト内の属性間の依存関係によって競合の定義が異なるため、アプリケーションごとに競合の定義を明確化する必要がある。

(2) 新規のオブジェクトに対するフェーズ

ここでは追加系のログの比較を行なうことによって、ディスコネクション中に新規に生成されたオブジェクトに対して競合を検出し、解消する。

ここで考慮しなければならないことが２点あるが、まず１点めは、複数のオブジェクト間でのアプリケーションのセマンティクス上の依存関係によって生じる競合の検出である。スケジューラを例にとって説明すると以下のようになる。

ディスコネクション中、クライアント側では 12 月 5 日の午前 10 時から顧客Ａ社への出張をスケジュールとして追加したのに対し、サーバ側では同日同時刻に課内ミーティングのスケジュールを追加した。この場合、アプリケーションのセマンティクス上２つのスケジュールは競合していることになる。このフェーズではこういった複数のオブジェクト間の競合も検出しなければならない。

もう１点めは、上記の例でも競合にならないケースもあるということである。前出の例では、クライアント側のユーザのスケジュールと、そのユーザが属するグループのスケジュールとの競合を示している。これに対して、クライアント側のユーザＡとサーバ側のユーザＢがそれぞれ追加した自分自身のスケジュールが同日同時刻であっても、競合とはならない。

このように複数のオブジェクト間でのアプリケーションのセマンティクス上の依存関係に基づく競合についても、明確に定義する必要がある。

(3) 既存と新規のオブジェクトに対するフェーズ

ここでは変更・削除系のログと追加系のログの比較を行なうことによって競合を検出し、解消する。具体的には、１つはクライアント側の変更・削除系ログとサーバ側の追加系ログとの間、もう１つはサーバ側の変更・削除系ログとクライアント側の追加系ログとの間で行なわれる。

ここで考慮すべきことは、新規オブジェクトに対するフェーズで言及したことと同じである。

4.2.5 競合解消と戦略

競合を検出した結果、それをある規則に基づいて解消する必要がある。その規則が競合解消戦略である。競合解消戦略の指定により、ユーザの介入負担を軽減するとともに、楽

観的並行制御におけるシリアライザビリティの保証度を高くすることも可能となる。

以下にそれぞれの競合解消戦略について説明する。

(1) クライアント優先

回線を切断していたクライアント側のユーザの変更を優先する。

(2) サーバ優先

サーバ側に接続しているユーザの変更を優先する。

(3) ユーザ優先度

各ユーザに付与された優先度に基づき、高い優先度をもつユーザの変更を優先する。回線を切断していたクライアント側のユーザが１人、サーバ側に接続しているユーザも１人の場合は、クライアント優先・サーバ優先の戦略と同じになる。

(4) 更新時間優先

それぞれの更新操作ログに対してタイムスタンプを付与する。更新時間がより最近（あるいは最も古い）の変更を優先する。現行では更新ログにタイムスタンプを付与しておらず、かつログのマージを行なっているので、サポートする場合はこの辺りの実装を修正する必要がある。また、ディスコネクション中はそれぞれが異なる内部時計を使用しているため、再接続の際の時刻調整が不可欠である。

(5) ユーザ介入

回線を切断していたユーザが競合解消時に自らインタラクティブに介入して競合を解消する。そのため両者の相違をユーザに示し、ユーザがどちらかを選択する、あるいは新しいデータを入力できるようにする必要がある。ここでは、競合したオブジェクトだけを示すのではなく、そのオブジェクトがアプリケーション全体の中でどのような位置付けにあるかなど、アプリケーションのセマンティクスに関わる情報も必要となる。それゆえ表示を支援する場合、アプリケーションからそうした情報を獲得するためのフレームワーク、記述形式等を考える必要がある。

(6) コーディネータ介入

ユーザ介入戦略の実行もここに含まれ、コーディネータモジュールが競合解消の支援を行なう。この戦略は付加機能であり、上記の戦略とともに使用される。競合解

消の結果、回線を切断していたクライアント側の変更が採用された場合、他のユーザとの交渉をサポートする。具体的には、現在使用中の他のユーザに対して、クライアント側の変更を通知し、他のユーザが受け入れられない場合はその旨を返答する。お互いに交渉することによって、変更を採用するか却下するかを決定する。また、現在使用中でない他のユーザに対しては、交渉結果をメールで通知する。

次に戦略の指定方法について、留意すべきことを以下に掲げる。

- サーバ側とクライアント側で、常に同一の戦略を保持していなければならない。
- あるユーザがディスコネクティッドオペレーションを実行している最中は、現行の戦略を変更することはできない。ディスコネクティッドオペレーションを実行しているユーザが存在しない時に限り、戦略を変更することができる。
- 競合解消中に戦略を変更することはできない。
- オブジェクトやグループ単位での戦略指定は一部制限される。それは以下に示すように、競合の定義に関する一貫性が崩壊する可能性がある。

  例えば、クライアント側がディスコネクション中にA、B２つのスケジュールを変更したとする。サーバ側ではAを変更するが、これがクライアント側のBの変更と競合する場合がある。この時、競合解消においてAではサーバ優先の戦略を、Bではクライアント優先の戦略を指定していると、新たに更新されたAとBは互いに競合する結果となる。

最後にもう１つ戦略について考慮すべきことを以下に掲げる。

それは、クライアント側でオブジェクトAを変更したのに対して、サーバ側では何も変更しなかった場合の扱い方である。上記に掲げた戦略は、どちらの変更を優先するかに重点が置かれており、片方が何も変更していない場合の扱いが曖昧になっている。サーバ優先の戦略で、サーバ側の変更がないオブジェクトに対し、元のサーバ側のオブジェクトに戻すのか、あるいは変更に重点を置いてクライアント側の変更を選択するのかについて、決めておかなくてはいけない。

4.2.6　結果ログのリプレイ

競合解消の結果を元に、オリジナルサーバと複製サーバの両者のデータをそれぞれ変更し、お互いの一貫性を維持するのがリプレイの手続きである。

競合解消の結果は、オリジナルサーバの場合ＤＯマネージャから、複製サーバの場合はＤＯマネージャからクライアント側にあるＤＯマネージャを経由して送られる。

ここで考慮しなければならないことは、サーバ内部のオブジェクトの参照関係を変更する必要があるということである。このオブジェクトの参照関係はアプリケーションのセマンティクスに依存するものであり、更新ログの作成時に考慮した操作に伴なう他オブジェクトの変更ログだけでは対処しきれないものである。

スケジューラを例にとると、次のような時このような問題が発生する。

> 各スケジュールオブジェクトは通常日時をキーにしてソートされている場合が殆んどである。そのため、ある１つのスケジュールを変更した場合、全スケジュールの順序を内部的に変更しなければならない時がある。この再ソートのような手続きは、アプリケーションのセマンティクスに依存するものであり、アプリケーションの内部仕様に関する情報なくしては実現できない。

次に考慮すべきことは、オリジナルサーバと複製サーバに対して並行にリプレイすることはできないということである。なぜなら、追加系操作によってクライアント側が生成した新規オブジェクトのＩＤは暫定的なものすぎないからである。正式なオブジェクトＩＤはオリジナルサーバ上でリプレイした結果得られるものであり、暫定的なＩＤと同一のものであるという保証は全くない。そのため、オリジナルサーバをリプレイし、正式なオブジェクトＩＤを得た時点でその旨を複製サーバのリプレイに反映させなければならない。

第 5 章

アプリケーション例

5.1 グループスケジューラ

この章では、ディスコネクティッドオペレーションをサポートするツールキットを利用したアプリケーションの一例として、グループスケジューラについて説明する。

5.1.1 グループスケジューラの概要

図 5.1は、ある会社の営業部で部員及び部全体の予定を管理するスケジューラの構成を示している。この図では部員Aが自分の移動計算機上にサーバアプリケーションの複製を保持し、社外でディスコネクティッドオペレーションを実行している。部長及び他の部員は、社内でオリジナルのサーバアプリケーションをアクセスしている。

図 5.2は、ＣＵＩのクライアントアプリケーションを用いて営業部、及び部員のスケジュールを実際に表示したものである。

このグループスケジューラは、サーバの部分はＣ＋＋、クライアントの部分やＩＰＣなどはＣ言語により実装されている。１つのスケジュールイベントがＣ＋＋の１つのオブジェクトに対応し、オブジェクトを識別するためのオブジェクトＩＤ（例：[oid = 16]）が各スケジュールイベントごとに割り当てられている。また、図 5.2の表示例では、各個人のスケジュールにグループ全体のスケジュールが含まれていないが、これは各スケジュールイベントのオブジェクトを簡単に出力しているためである。グループのスケジュールと各個人のスケジュールの競合に関しては内部的に考慮されている。

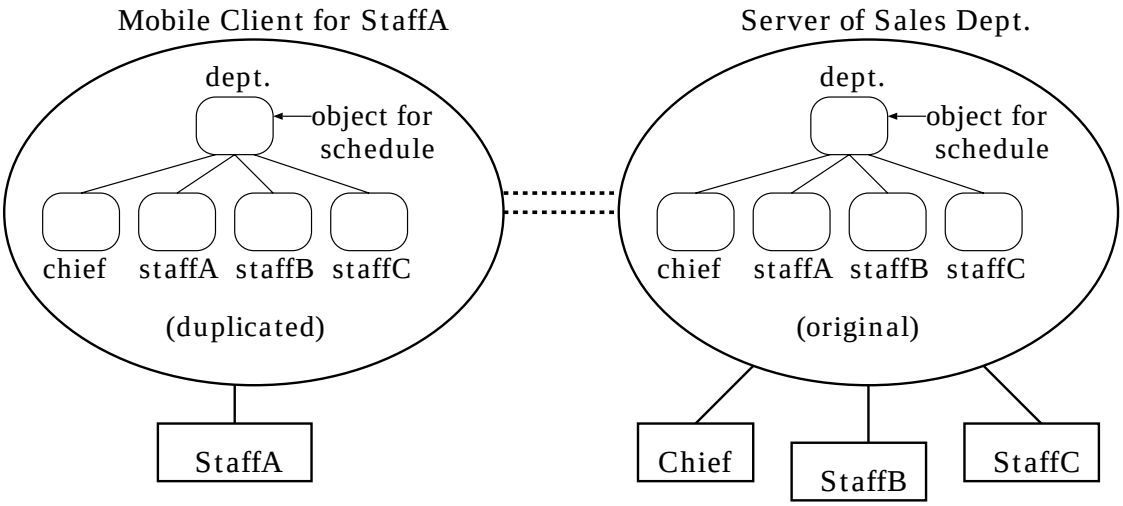


図 5.1: グループスケジューラの構成

```
command : sas
=== ShowAllSchedule ===
Input group name =>SalesDept
 ===  schedule : "SalesDept" group ===
  MeetingA    <  1> [oid =  16] at 1998/02/09/13:00 for   1hour(s)
  MeetingB    <  2> [oid =  15] at 1998/02/12/10:00 for   2hour(s)
 ===  schedule : "Chief" of "SalesDept" group ===
  CustomerA  <  7> [oid =  17] at 1998/02/09/15:00 for   2hour(s)
  HeadQuater<15> [oid =  18] at 1998/02/10/13:00 for   5hour(s)
  MeetingC    <  3> [oid =  19] at 1998/02/13/09:00 for   1hour(s)
 ===  schedule : "StaffA" of "SalesDept" group ===
  CustomerB  <  8> [oid =  20] at 1998/02/12/13:00 for   3hour(s)
 ==  schedule : "StaffB" of "SalesDept" group ===
  CustomerC  <  9> [oid =  22] at 1998/02/09/14:00 for   2hour(s)
  CompanyC  <  6> [oid =  21] at 1998/02/13/09:00 for   3hour(s)
 ==  schedule : "StaffC" of "SalesDept" group ===
  BankA        <10> [oid =  23] at 1998/02/10/10:00 for   2hour(s)

command :
```

図 5.2: グループスケジューラの表示例

5.1.2 トレードオフの考慮例

ここでは前記のグループスケジューラを用いて、通信コストと一貫性のトレードオフをユーザが考慮することの有効例を示す。

部員Aは出張の際、サーバにあるスケジューラを自分の移動計算機に複製としてダウンロードし、営業部のネットワークとの接続を切る。出張先で自分のスケジュールを変更する際、その更新が他の部員に影響を与えないならば、営業部のネットワークにすぐに接続する必要はない。帰社してからその更新をオリジナルのサーバに反映させればよい。このような場合、自分のスケジュールに対しては一貫性よりも通信コストを優先する設定を行なうことができる。これにより一貫性は帰社するまで回復されないが、ユーザの要求に従った通信コストの削減が実現される。

また、彼が出張先で営業部全体の会議を開きたいと考えた場合、スケジュールの変更が即座にオリジナルのサーバ側に反映されないと、その間に他の部員がその日時に別の予定を入れてしまう可能性がある。そのため、高い通信料を払ってでもすぐに営業部のネットワークに接続する必要がある。このような場合、部全体のスケジュールに対しては、通信コストよりも一貫性を優先する設定を行なうことができる。これにより通信コストは高くなるが、ユーザの要求に従った一貫性の維持が保証される。

第 6 章

評価

この章では、ディスコネクティッドオペレーションの機能において通信コストと一貫性のトレードオフを考慮することの有効性を実証する。使用するアプリケーションは、前章で述べたグループスケジューラである。

6.1 評価条件

評価は以下の条件のもとに行なわれた。

- サーバ側、回線切断しているクライアント側の両者で、グループスケジュールの変更を 16 回、それぞれ異なるスケジュールオブジェクトに対して実行する。
- ある一回の変更では、両者はほぼ同時期に同じオブジェクトＩＤをもつスケジュールオブジェクトを違った値で更新する。
- それぞれの変更操作で更新されるオブジェクトのデータ部の場所とサイズ (24byte) は毎回全く同じである。
- 競合解消においてはクライアント側の変更が優先され、他のスケジュールと競合する変更は行なわない。
- 決められた変更操作回数毎にサーバ側と通信して一貫性を維持し、その変更操作回数を変化させることにより、実際の接続間隔時間の変化をシミュレートする。

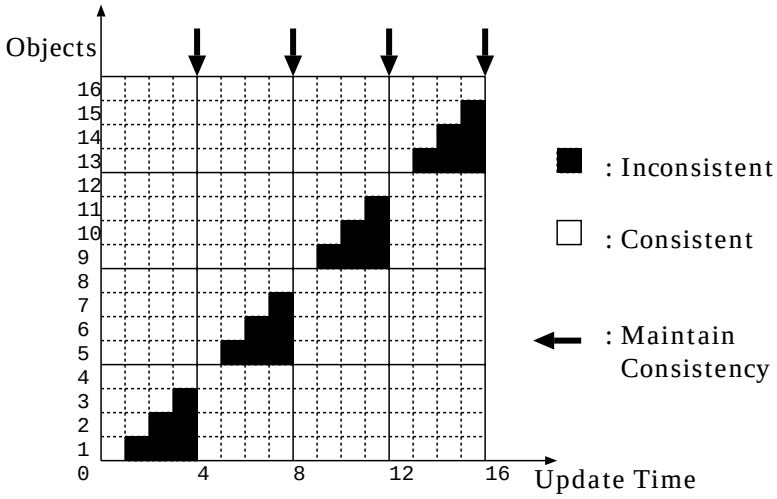


図 6.1: 一貫性維持度の算出方法

- サポートされている実装はマッピングの具体例の説明で使用された実装 I_{CD} であり、その種類は $I_{CD_1}, I_{CD_2}, I_{CD_3}$ の３種類である。（$E_{CD} = 0.80$）

- トレードオフの記述を実装に伝えるためのマッピングには、マッピングの具体例の説明で使用された $Interval(P_{cd})$ の式が用いられる。

- 測定される時間は、クライアント側が接続を要求してから、サーバ側からの一貫性維持終了の通知が来るまでであり、クライアント側での複製サーバの一貫性維持の時間は含まれない。（上記の条件では複製サーバの一貫性維持は不要）

- 一貫性維持度は単純化するために、下記のようにして算出した。

  図 6.1に示したのは、１６回の変更操作を横軸に、１６個のオブジェクトを縦軸にとったグラフである。これは、４回の変更操作ごとにサーバ側と接続して一貫性の維持を行なった場合の例であり、それぞれのオブジェクトがオリジナルのサーバと複製サーバの間で一貫性がとれているか否かを示している。

  一貫性維持度は、このグラフ全体に占める黒く塗り潰されていない部分の面積比として表わされる。

  よってこの例では 0.906 が一貫性維持度となる。

表 6.1: 測定結果

I_{CD}	P_{CD}	接続間隔時間 (分)	接続間隔毎の操作数	接続時間 (ms)	合計時間 (ms)	相対コスト	一貫性維持度
I_{CD_1}	-1.00	常時接続	—	—	—	∞	1
I_{CD_2}	-0.80	15	1	15.66	250.56	10.05	1
	-0.53	30	2	20.14	161.12	6.46	0.968
	0.00	60	4	21.00	84.00	3.37	0.906
	0.27	120	8	22.45	44.90	1.80	0.781
	0.80	240	16	24.91	24.91	1	0.531
I_{CD_3}	1.00	接続なし	—	0	0	0	0.531 以下

6.2 測定条件

測定に使用した計算機環境は、以下の通りである。

- サーバ側は、ＣＰＵが Pentium 100MHz、メモリが 32MB、ＯＳは RT-Mach。
- クライアント側は、ＣＰＵが PentiumMMX 133MHz、メモリが 64MB、ＯＳは RT-Mach。
- ネットワークは現状では 10BASE-T

  （近い将来、PHS や携帯電話などの公衆回線を使って測定する予定）

6.3 測定結果

表 6.1に測定結果を示す。

また、トレードオフを考慮することの有効性を明示するため、表 6.1 から得られた相対的な通信コストと一貫性維持度の相関関係を図 6.2に示す。

評価・測定条件を十分認識した上でここで得られた測定結果を解釈すると、次のようなことが伺える。

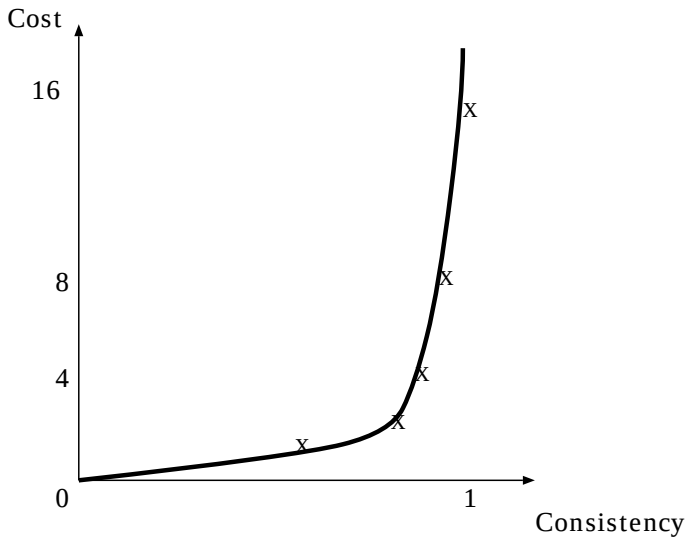


図 6.2: 通信コストと一貫性維持度の相関関係

- 一貫性維持への要求が高ければ通信コストが高くなる一方、一貫性維持への要求が低ければ通信コストは低くなる。

  グループスケジューラーのようなユーザの一貫性に対する要求が多様に変化するアプリケーションにおいては、通信コストと一貫性のトレードオフをユーザが考慮することが有効な手段となる。

- I_{CD_2}における相対的な通信コストは、ほぼサーバ側との接続回数に比例する。

  これはログを用いた一貫性維持、特に競合解消に伴なう処理のオーバーヘッドが小さいことを意味する。オブジェクトの総数が少なく、そのデータサイズも小さく、またディスコネクション中の更新頻度が小さいことがその理由である。しかしながら、小人数で利用するグループスケジューラをアプリケーションとして利用し、評価を実施していることを考慮すれば、オブジェクトの総数や更新頻度が急激に増大することがないため、妥当な結果ではないかと思われる。

- 一貫性維持度が高い時にさらに維持度を上げようとした場合、その通信コストは維持度が低い時に比べてかなり高い。

第 7 章

議論

7.1 トレードオフを記述するインタフェース

トレードオフを記述する汎用的なインタフェースを実現する上で、以下のような問題が挙げられる。

(1) 各メトリックスの尺度に対する明確な定義

今回の評価の際に使用した一貫性維持度は、一貫性というメトリックの尺度である。この尺度を単純に定義することにより評価を行なったが、この尺度に対する一般的な定義はまだまだ曖昧である。また、通信コストについても同様で、現実には使用する公衆回線によって初期接続料金や接続時間当たりの料金も多種多様であり、常に不変の要因ではない。

このように各メトリック、特にその尺度について明確に定義することは、メトリックス、アプリケーション、ユーザの使い方、インフラストラクチャーなどによってそれぞれ異なるため、非常に困難な作業である。

(2) トレードオフ記述の困難さ

簡単なシンタクスのインタフェースを提供したとしても、ユーザにとって実際にトレードオフを記述するのは非常に難しい。これは上記した定義の曖昧性に大いに依存する。ユーザは実際に使っていく上で、徐々にチューニングをしていくのが現実であろう。しかしながら、その負担は大きく、彼らに高度な知識を要求することになり、簡単なシンタクスも複雑すぎるセマンティクスゆえに敬遠されるであろう。

それゆえ、このような負担を解消、軽減する策を講じる必要がある。システムをサポートする側が、ケーススタディを通して各メトリックスのパラメタ値の設定目安を示すことが、１つの解決策として考えられる。しかしながら、それではシステマティックな対応ではない。サポートするシステムやトレードオフの種類に依存せずに、この設定目安を示すことができるようなフレームワークを考え、ツール提供の形でシステマティックに対応する方向を考えるべきである。

(3) 有効性の実証

トレードオフを記述する汎用インタフェースの有効性を実証するには、様々なシステムにおける各種のトレードオフについての効果を検証することが不可欠である。今回実証した通信コストと一貫性のトレードオフ考慮の有効性は、１つのケーススタディとして得られた結果にすぎず、汎用インタフェースの提供に向けて第一歩を踏み出したにすぎない。

ケーススタディで得られた教訓を、インタフェースの定義やマッピングのメカニズムにフィードバックし、それに基づいてケーススタディを繰り返すことが重要である。

7.2 ディスコネクティッドオペレーションのツールキット化

オブジェクト指向のアプリケーションに対して、ディスコネクティッドオペレーションの機能を透過的なツールキットとして提供するには、以下に掲げるような問題を解決する必要がある。

(1) アプリケーションのセマンティクスに依存する競合

競合検出のところでも述べたように、同一のオブジェクト内の異なる位置の競合、複数のオブジェクト間の競合については、アプリケーションのセマンティクスに依存する競合であるため、システム側では正確に競合を検出できない場合がある。そのため、セマンティクスに関する情報をアプリケーション側から提供してもらう必要がある。それを実現するためには、様々なアプリケーションに対応した競合を明確に定義し、その定義に基づいた記述形式を用意する必要がある。しかしながら、アプリケーションのセマンティクス関わる部分を厳密に定義することは非常に難しい。

(2) 競合解消戦略による楽観的並行制御におけるシリアライザビリティの保証

競合解消の手続きを自動化するために、いくつかの競合解消戦略を提案したが、それぞれの戦略がユーザの意図した通りに正しく動くかどうかは十分に検証されていない。それらを検証するためには、それぞれの戦略がどれくらいの精度で、楽観的並行制御におけるシリアライザビリティを保証するかどうか調べてみることが必要となる。しかしながら、現実には非常に難しい検証となることは確かである。

(3) 更新ログ作成とそのタイミング

サーバアプリケーション内のオブジェクトの変更を捕捉し、更新ログを書き出すタイミングは、アプリケーションに依存して行なっているのが現状である。すなわち、あるオブジェクトの変更を行なった際、その直後に更新ログを作成する手続きを明示的に呼び出している。そのため透過的なツールキットを提供する上では非常に大きな問題となっている。

この問題を解決するためには、言語処理系によるサポートが欠かせないが、そのサポートなしで実現するには、アプリケーションの大幅な変更を余儀なくされるため、アプリケーションプログラマにとっては大きな負担にならざるを得ない。

(4) オブジェクトの識別

ディスコネクティッドオペレーションの実現においては、オリジナルサーバと複製サーバ間でオブジェクトを同定するために、オブジェクト識別子が必要となる。Ｃ＋＋やＪＡＶＡのようなオブジェクト指向型のプログラミング言語では、通常ユーザがオブジェクトの識別を意識する必要がないので、オブジェクト識別の機能が提供されていない。

今回の実装では、オブジェクト識別の機能をオブジェクトＩＤ発行モジュールとしてクラスライブラリの形で用意した。これを使用することにより、ユーザはオブジェクトの識別を意識することなくディスコネクティッドオペレーションの機能を利用できる。しかしながら、パーシステントシステムのようにオブジェクトの識別を必要とする他のシステムサポートを考慮すれば、共通に利用できる統合されたオブジェクト識別機能を用意すべきである。

第 8 章

関連研究

ディスコネクティッドオペレーションは CMU の Kistler ら [1] によって提唱された。これは分散ファイルシステムである Coda[2] 上で実装されており、必要とされるファイルを予めプリフェッチすることによって、ディスコネクティッドオペレーションを実現している。

Coda の前進である AFS[3] に対し、ディスコネクティッドオペレーションをサポートした [4] のが、Little Work Project[5] である。Little Work では、モーバイルコンピューティング環境を意識し、バンド幅の小さな通信回線上での効率的なファイルの送受信を実現している [6]。また、モーバイルファイルシステムとして、接続 (Coneected)・部分接続 (Partially Connected) ・フェッチのみ (Fetch-Only)・切断 (Disconnection) の４種類のモードを切替えることによって、キャッシュミスによる実行停止を避け、かつ通信回線の利用を最小限におさえている [7]。

その他にバンド幅の小さな通信メディア上で効率的なファイルの送受信を実現している研究として、NFS 上でのツールキットを作成した Duchamp の研究 [8] などがある。

また、Odessey では、UNIX をモーバイル用に拡張した OS 上に、Coda で実現したディスコネクティッドオペレーションの機能を実装している [9][10]。

ファイルではなく、オブジェクト指向のアプリケーションに対して、ディスコネクティッドオペレーションをサポートするツールキットを提供しているのが、Rover Tool Kit[11][12] である。Rover では、複数の RPC を１つの RPC にまとめる QRPC(Queued RPC) の機構を使用し、バンド幅の小さな通信メディア上での効率的なデータの送受信を実現している。しかしながら、どのオブジェクトをプリフェッチするかについては全く述べられてい

ない。

これまで述べてきたディスコネクティッドオペレーションに関する研究では、ユーザの要求に基づいて複数あるメトリックス間のトレードオフを考慮するようなことは行なわれていない。

また、ユーザがトレードオフを記述するインタフェースを定義し、それを実装に伝えるためのメカニズムを提供している研究は、他には見当たらない。

オブジェクトをグループ化して扱っている研究としては、Java を拡張した Maryland 大の Sumatra[13] がある。また、Odessey には、ファイルのグループ化によってプリフェッチのヒット率を向上させる Dynamic Sets の機能がある。

一貫性の維持に関しては、上記のようなディスコネクティッドオペレーションを実現するほとんどのシステムにおいて、更新ログを利用した楽観的並行制御 [14] を採用している。Coda File System の First-Class Replication では悲観的並行制御を採用しているが、これは分散ファイルシステムの機能としてである。

楽観的並行制御における競合解消の問題については、上で述べられたファイルを利用したシステムを含めて様々な研究がなされており、更新ログの最適化に関する研究 [15] なども行なわれている。

しかしながら、アプリケーションのセマンティクスへの依存性が非常に高いオブジェクト指向のアプリケーションでの競合解消については、あまり研究されていない。前出の Rover などのように、競合解消ができない時はユーザに判断させるとしてしている研究が殆んどであり、競合解消戦略を導入している研究は見当たらない。

オブジェクトの識別機能としては、C++のパーシステントオブジェクトをサポートする Texas[16] が採用している Type Descriptor などがある。

第 9 章

おわりに

9.1 結論

本研究では、適応可能なアプリケーションの構築を実現するために、ユーザがトレードオフを記述するインタフェースを提唱した。インタフェースの記述仕様を示すとともに、実装へのマッピング方法についてのフレームワークやメカニズムについても提唱した。

トレードオフを記述するインタフェースの有効性を示すために、検証例としてディスコネクティッドオペレーションの機能をサポートするツールキットの設計、実装を行なった。実際の評価としては、通信コストと共有データの一貫性とのトレードオフを例に挙げ、各メトリックのパラメタ値を変化させ、それぞれの相対的な通信コストと一貫性維持度の比を求めた。その結果、相対的な通信コストと一貫性維持度の相関関係が、パラメタ値を反映したトレードオフの関係にあることが実証された。

設計・実装の面では、上記のディスコネクティッドオペレーションの機能をオブジェクト指向のアプリケーションに提供するツールキットの作成を試みた。また、アプリケーション例としてグループスケジューラを実装した。

汎用的・透過的なツールキットの提供に際しては、アプリケーションのセマンティクスへの依存性、競合解消戦略と楽観的並行制御におけるシリアライザビリティの保証度の関係、といった様々な問題点を提起することができた。

9.2 今後の課題

今後の最も重要な課題は、トレードオフを記述するインタフェースの有効性を示すために、さらなる検証を繰り返すことである。メトリックスの厳密な定義をもとに様々なトレードオフや実装を試み、それぞれの検証結果をインタフェースの定義にフィードバックさせることによって、経験的実証に基づくシステマティックなインタフェースの記述仕様を目指していきたい。

また、さらなる適応可能なアプリケーション構築を実現するため、計算機に関する環境情報を利用してマッピングレイヤーを変えるメカニズムを実現していきたい。これにより、ユーザやアプリケーション側からだけでなく、計算機システム側からの情報も利用できるようになる。こうした両方向からの情報を統合的に扱える仕組みをどのように提供するかについても考えていきたい。

その他、ディスコネクティッドオペレーションのツールキット化に関して提起されたアプリケーションのセマンティクスへの依存性の問題などを、システマティックに解決する仕組みを実現していきたい。その際、決してディスコネクティッドオペレーションの機能にこだわらず、パーシステントシステムやオブジェクトのネーミングサービスなどオブジェクト指向型のアプリケーションに対するシステムサポートという観点からより汎用的なフレームワークについて考えていきたい。

参考文献

[1] J. J. Kistler, and M. Satyanarayanan, "Disconnected Operation in Coda File System," *ACM Transactions on Computer Systems, Vol.10, No.1*, pp.3-25, February 1992.

[2] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers, Vol.39, No.4*, April 1990.

[3] J. H. Howard, "Overview of the Andrew File System," *USENIX Winter Technical Conference*, February 1988.

[4] L. B. Huston, and P. Honeyman, "Disconnected Operation for AFS," *USENIX Symposium on Mobile and Location-Independent Computing*, pp.1-10, August 1993.

[5] P. Honeyman, L. B. Huston, J. Rees, and D. Bachmann, "The LITTLEWORK Project," *Third IEEE Workshop on Workstation Operating Systems*, April 1992.

[6] L. B. Huston, and P. Honeyman, "Partially Connected Operation," *Second USENIX Symposium on Mobile and Location-Independent Computing*, pp.91-97, April 1995.

[7] P. Honeyman, and L. B. Huston, "Communications and Consistency in Mobile File Systems," *IEEE Personal Communications : Special Issue on Mobile Computing*, October 1995.

[8] D. Duchamp, "A Toolkit Approach to Partially Connected Operation," *USENIX 1997 Annual Technical Conference and Networking*, January 1997.

[9] M. Satyanarayanan, "Mobile Information Access," *IEEE Personal Communications, Vol.3, No.1*, February 1996.

[10] M. Satyanarayanan, "Fundamental Challenges in Mobile Computing," *Fifteenth ACM Symposium on Principles of Distributed Computing*, May 1996.

[11] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek, "Mobile Computing with the Rover Toolkit," *IEEE Transactions on Computers : Special issue on Mobile Computing*, February 1997.

[12] A. D. Joseph, and M. F. Kaashoek, "Building Mobile-Aware Applications using the Rover Toolkit," *2nd ACM International Conference on Mobile Computing and Networking*, November 1996.

[13] M. Ranganathan, A, Acharya, S. D. Sharma, and J. Saltz "Network-aware Mobile Programs," *USENIX 1997 Annual Technical Conference and Networking*, January 1997.

[14] H. T. Kung, and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems, Vol.6, No.2* June 1981.

[15] L. B. Huston, and P. Honeyman, "Peephole Log Optimization," *IEEE Workshop on Mobile Computing Systems and Applications*, December 1994.

[16] V. Singhal, S. V. Kakkad, and P. R. Wilson, "Texas: An Efficient, Portable Persistent Store," *Fifth International Workshop on Persistent Object System*, pp.11-33, September, 1992.

[17] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System," *1st USENIX Mach Symposium*, October, 1990.

[18] 小林 勝, 会津 宏幸, 嶋本 堅司, 中島 達夫, "環境サーバの設計と実装," 第 *14* 回日本ソフトウェア科学会全国大会, October 1997.