平成15年度学士論文

ソケットの拡張による Java用分散ミドルウエアの 高信頼化

東京工業大学 理学部 情報科学科

学籍番号 00-2205-9

松田麻里

指導教官

千葉 滋 助教授

平成16年2月5日

概要

今日、ノート PC などの携帯情報端末を用いて、場所を選ばずにネットワークを利用することが可能である。このようなモバイルコンピューティングの環境では、ユーザーは場所を移動するたびにネットワークを切断し、移動先で新しいネットワークにつなぎ直さなければならない。また無線で通信をしている場合は、場所の移動によって IP アドレスが変更されることがある。

このような、ネットワークの物理的な切断や IP アドレスの変更が行われると TCP コネクションは失われてしまう。分散アプリケーションなどを使用して、リモートホストと通信を行っているような状況では、移動するたびに通信は途切れてしまうことになる。そのため移動先で分散アプリケーションを起動し直さなければならない。しかし、ユーザーは移動先でも分散アプリケーションを続けて使用したいと思うだろう。

ネットワーク切断後のコネクションの回復という処理をアプリケーションが行うのは困難であり、こうした処理はアプリケーションに対して透過的に行うことが望ましい。また、アプリケーションの下位に位置して、アプリケーションに対して基本的な処理を行うミドルウエアを変更するという方法もある。しかしこの方法では、現在多数存在する各分散ミドルウエアの実装を変更しなければならず汎用性が乏しくなる。

そこで、分散ミドルウエアを高信頼化するために、ミドルウエアの下位に位置するソケットを高信頼化することを提案する (ReliableSocket)。高信頼化の処理をソケットレベルで行うことで、様々なミドルウエアで使用することができ、ミドルウエアの実装の変更などの影響を受けにくい。またミドルウエアを利用しない分散アプリケーションにも対応することができる。

ReliableSocket は TCP コネクションの情報を保存し、ネットワーク切断時の例外を検出すると保存してある情報をもとに自動的に新しいコネクションを作り直す。そしてネットワーク切断時に失われたデータを再送する。また、再接続してきた相手が今まで通信していた相手であるかどうかを確かめるために認証を行う。

本研究では、ReliableSocket を JavaRMI に組み込むことで、JavaRMI を高信頼化した。JavaRMI はデフォルトでは Java.net.SocketAPI を使って通

信を行っている。しかしソケットの差し換え機能があるので、Java.net.Socket の代わりに ReliableSocket を使うように変更することができる。

JavaRMI を使って、ネットワーク越で音楽を配信するアプリケーションを作成してデモを行った。そして、ネットワークを何度切断してもしばらく時間をおいてつなぎ直したら音楽が続きから再生できることを確かめた。

実験では ReliableSocket のスループットと遅延時間を測定した。スループットは送信するデータのサイズが数十バイト程度のときは標準の Java.net.Socket の十分の一位になってしまう。しかし、送信データのサイズを数百バイト程度にすれば Java.net.Socket の 3 分の 2 位になった。遅延時間は Java.net.Socket に比べて 0.5msec 程度大きくなった。

謝辞

本研究を進めるにあたり、研究の方向付けや論文の組み立て方についての助言をして頂いた指導教官の千葉滋先生に大変感謝致します。

同研究室の光来健一先生にはシステムの設計・実装そして実験をする上で多くの有益な助言を頂きました。大変感謝致します。

同研究室の佐藤芳樹氏、宇崎央泰氏、栗田亮氏、中川清志氏、西沢無我氏、須永豊氏、松沼正浩氏、柳沢佳里氏にも多くの助言を頂きました。

以上の方々に重ねて御礼を申し上げます。

目 次

図 目 次

表 目 次

第1章　はじめに

今日、ノート PC や PDA(Personal Digital Assistant) などの携帯情報端末を用いて、時間と場所を選ばすにネットワークを利用することができる。このようなモバイルコンピューティングの環境では、ユーザーは場所を移動するたびにネットワークを切断し、移動先のネットワークにつなぎ直さなければならない。無線 LAN で通信をしている場合は、場所の移動によって IP アドレスが変更されることがある。

このようなネットワークの物理的な切断や IP アドレスの変更が行われると TCP コネクションが失われてしまう。分散アプリケーションを使用してリモートホストと通信を行っているような状況では、移動するたびに通信が途切れてしまうことになう。このため現状では移動時にユーザ側が分散アプリケーションを中断し、移動先で起動し直さなければならない。しかしユーザーはアプリケーションの明示的な中断・再開処理をせずに、移動先でも続けてアプリケーションを使用したいと思うだろう。また、ネットワーク切断時には送信中のデータが失われてしまうこともある。そのため、移動先でアプリケーションを起動し直しても、一部のデータは失われている可能性もあり、続きから通信を再開できるとは限らない。

一度失われたコネクションは何らかの処理を行わない限り回復することはできない。しかしながら、このようなコネクションの回復という処理をアプリケーションが行うは困難であり、アプリケーションに対して透過的に行うことが望ましい。アプリケーションの下位に位置するミドルウエアを変更するという方法もある。しかしこの方法では、現在多数実用化されている各分散ミドルウエアの実装を変更しなければならず、汎用性が乏しくなる。

そこで我々は、分散ミドルウエアを高信頼化するためにミドルウエアの下位に位置するソケットを高信頼化することを提案する。高信頼化という処理をソケットレイヤで行うことにはいくつかの利点がある。まず、様々なミドルウエアを容易に高信頼化することができる。これは、ミドルウエア内部で使用しているソケットを高信頼ソケットと置き換えるだけで済むからである。また、ミドルウエアの実装が変更されたとしても影響を受けなくてすむ。さらに、カーネルやネットワークプロトコルの変更を必要としない。

高信頼化とはコネクションが失われてもネットワークに再接続すれば続きから通信を再開できるようにすることである。高信頼化したソケット (ReliableSocket) は java.net.Socket に以下のような機能を追加した。

- 通信中の TCP コネクションの情報（サーバの IP アドレスと再接続専用のポート番号、送受信バッファの内容）を保存する
- コネクションが失われたら自動的にサーバに再接続する。そしてネットワークに再接続したら、保存している情報をもとに新しいコネクションを張る
- 再接続してきた相手が今まで通信をしていた相手であるかを確かめるために認証を行う
- 切断時に失われたデータを回復し、通信が続きから再開できるようにする

分散ミドルウエア内で使用するソケットを ReliableSocket と置き換えることで分散ミドルウエアを高信頼化することができる。本研究では実際に JavaRMI に ReliableSocket を組み込んで高信頼化できていることを確かめた。JavaRMI は現在広く使われている分散ミドルウエアの一つであり、デフォルトでは java.net.Socket を使っているがカスタム RMI ソケットファクトリを作成することでソケットを差し換えることができる。

本稿の残りは、次のような構成からなっている。第 2 章は モバイルコンピューティングの説明と関連する技術について、第 3 章では、システムの実装、第 4 章では、実験結果を、そして第 5 章でまとめを述べる。

第2章　モバイルコンピューティングについて

この章ではモバイルコンピューティング [?] に関する説明を行い、その実現のために必要な技術に関する考察を行う。

2.1　モバイルコンピューティングとは

モバイルとは「動くことが可能な」、「動くことができる」といういい意味を表し、コンピューティングとは「コンピュータを利用すること」を意味する。つまり、モバイルコンピューティングとは「一箇所に固定されずに、移動しながらコンピュータを利用すること」と解釈することが可能である。
モバイルコンピューティングが普及した背景にはハードウエアや情報インフラストラクチャの進歩ががあげられる。PC、特にノートPCやPDA(Personal Digital Assistant) に代表される携帯情報端末の小型軽量化によって、どこでもコンピュータを持ち運んで利用することが可能となった。また、情報インフラストラクチャの進歩により、形態情報端末を使って、外出先でネットワークを利用することも可能である。最近話題となっているホットスポットとは無線LANでインターネットに接続するためのアクセスポイントが設置されている場所のことで、駅や空港、ホテル、喫茶店・ファーストフード店あるいは街角など様々なところに用意され始めている。
このような背景から、「モバイルコンピューティング」という言葉には「移動しながらコンピュータを利用する」という意味だけでなく「移動しながら通信によるコミュニケーションを行う」という意味も加えられるようになっている。モバイルコンピューティングの環境では、外出先でも家や職場と同様の環境を再現できるようになりつつある。

2.2　モバイルコンピューティングに必要な性質

モバイルコンピューティングの環境では、場所を移動しても分散アプリケーションを続けて使用できることが望ましい。例として以下のような状

況を考える。

あるユーザーが外出先でリモートホスト上のエディタを手元のノート PC に表示させて利用している。しかし都合により場所を移動しなくてはならなくなった。そのため、いったんノート PC を外出先のネットワークから切断し、移動先で再びネットワークに接続したらエディタを続けて利用することができた。

このような状況はモバイルコンピューティングの環境では非常に多く起るだろうと考えられる。しかし、現状ではネットワークが切断さると TCP コネクションは失われてしまい、その結果、分散アプリケーション（上の例ではエディタ）は中断してしまう。無線で通信をしている場合でも場所の移動によって IP アドレスが変更されると TCP コネクションは失われてしまう。一度失われたコネクションは何らかの処理を行わない限りは回復することはできないので、移動先でネットワークに再接続してもアプリケーションを続けて使うことはできない。そのため、上の例のような状況では、移動時にユーザ側がアプリケーションの中断処理を行って、移動先で再開処理を行わなければならない。しかし、モバイルコンピューティングという環境を考えると、このような中断・再開処理の部分は透過適に処理されるべきである。また、ネットワーク切断時には送信中のデータが失われてしまうこともあるので、移動先でアプリケーションを起動し直しても一部のデータは受信でいない可能性もある。

本研究ではこのような問題を解決するために高信頼ソケットを開発した。高信頼ソケットはネットワークの物理的な切断や IP アドレスの変更後も、ネットワークに再接続したらコネクションの回復と切断時に失われたデータの回復を行い通信を続きから再開できるようにする。この拡張したソケットを分散ミドルウエアに組み込むことで、分散ミドルウエアを容易に高信頼化することが可能である。

2.3　関連研究

2.3.1　TCP

TCP/IP

異なるネットワーク間で通信を行うためには、通信のための手順（プロトコル）を統一する必要がある。TCP/IP[**?**] とは Transimission Control Protocol/Internet Protocol の略で、インターネットで一般に使用されているプロトコルである。TCP/IP は UNIX には標準で採用されており、インターネット上のプロトコルとして、事実上の標準となっている。TCP/IP は TCP と IP という二つのプロトコルを意味する場合もあるが、一般的

には IP を利用したり、IP で通信をするときに必要となる各種プロトコルも含めて TCP/IP と呼ぶことが多い。

TCP/IP の構成

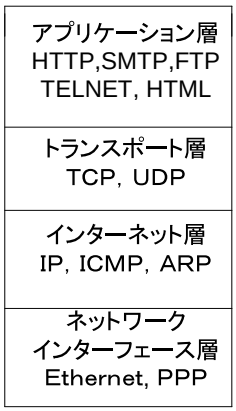

図 2.1: TCP/IP の階層モデル

TCP/IP は図 2.1 のような階層をなしており、TCP/IP はアプリケーション層、トランスポート層、インターネット層、ネットワークインターフェース層の 4 つの層から構成される。データを送るときは、高い層から低い層にデータが渡されてそれぞれの層で通信制御情報が付け加えられる。データを受けるときは、低い層から高い層にデータが渡されて、それぞれの層で必要な制御情報を読み取ったら取り除かれていくという仕組みになっている。

(1) ネットワークインターフェース層

物理的に接続されたコンピュータ間のデータ通信を行う。情報を実際に伝送するためには、同軸ケーブルやツイストペアケーブル、光ファイバケーブルなど、ネットワークに応じた物理的な伝送媒体が必要となる。また、これらさまざまな伝送媒体を介して通信を行うためには、端末やコンピュータと伝送媒体との接続に関するインターフェースを決めておく必要もある。こうした取り決めを行って、最終的にネットワークを介したデータ通信を実現するのがネットワークインターフェース層の役目である。代表的なプロトコルは PPP（Point to Point Protocol), イーサネットなどである。電話回線では PPP 用いてダイヤルアップ IP 接続を行う。また、

LAN への接続時には Ethernet(イーサネット) を用いる。

(2) インターネット層

最終的にデータのやりとりをするノード同士の通信を担当する。相手のノードを特定するために使われるのが IP アドレスである。そして最終的にデータのやりとりをするノードまでの道筋を決める。これをルーティングという。プロトコルには IP,ICMP,APR がある。

- IP(Internet Protocol)

  ネットワークに参加している機器の住所付け（アドレッシング）や、相互に接続された複数のネットワーク内での通信経路の選定（ルーティング）をするための方法を定義し、インターネット全体にパケットを送り届ける。IP アドレスはインターネットやイントラネットなどの IP ネットワークに接続されたコンピューター台一台にに割り振られた識別番号である。IP はデータリンクの特性を隠す役割もあり、通信したいホストの間の経路がどのようなデータリンクになっていようとも通信を可能にする。

- ICMP(Internet Control Message Protocol)

  IP によるデータ通信をサポートするプロトコルで、相手先のノードにデータを届けることができない場合にメッセージを出したり、ルーティングによって決められたデータの経路を変更する指示を出したりする。

- ARP(Address Resolution Protocol)

  インターネット層と、ネットワークインターフェース層の中間に位置して、IP アドレスと MAC アドレスの変換を行う。

(3) トランスポート層

アプリケーション間の通信を実現する。コンピュータでは複数のプログラムを同時に動作させることができるが、どのプログラムとどのプログラムが通信しているかを識別するのはトランスポート層の役割である。プログラムを識別するにはポート番号と呼ばれるアドレスが使われる。TCP/IP には TCP と UDP の二つのトランスポートプロトコルがある。

- TCP(Transmission Control Protocol)

  コネクション型の信頼性のあるプロトコルで、エンドホスト間でのデータの到達を保証する。コネクション型とはデータの受信側/送信側で確認をとりながら通信を行うことである。TCP は経路の途中でデータがなくなったり、順番が入れ替わるのを防ぐ機能やネットワークの利用効率を向上させるための複雑な機能を持つ。この結

果、TCP は非常に信頼性が高い分スピードは遅くなるのでメールや Web ページのデータなど確実に相手に届く必要があるデータの通信には向いているが、多少のデータ消失がおきても問題ないストリーミングには向いていない。

- UDP(User Datagram Protocol)

  コネクションレス型で、信頼性のないプロトコルである。データが相手に届かない場合や、相手のコンピュータがネットワークに接続されていないといったケースの場合、UDP は相手からの応答の確認などといった、送信したデータが届いているかどうかのチェックはしない。そのため、UDP を使う場合、そうしたチェックはアプリケーションプログラムが必要に応じて行うことになる。UDP はデータの量が少ない場合や、ブロードキャストやマルチキャストの通信、ビデオや音声などのマルチメディア通信に向いている。

(4) アプリケーション層
ネットワークを通じてやりとりしたデータを実際に利用し、ユーザーにサービスを提供する役割を持っている。SMTP, FTP, TELNET,POP, DHCP, SNMP などのプロトコルがこの層に属している。メールソフトやブラウザなどのネットワーク関連のアプリケーションはこれらのプロトコルに対応している。

- SMTP(Simple Mail Transfer Protocol)

  電子メールを転送する。

- FTP(File Transfer Protocol)

  ファイルを転送する。

- TELNET

  ほかのネットワークに遠隔ログインする。

- POP(Post Office Protocol)

  電子メールを保存しているサーバからメールを受信する。

- DHCP(Dynamic Host Configuration Protocol)

  インターネットに一時的に接続するコンピュータに、IP アドレスなど必要な情報を自動的に割り当てる。

- SNMP(Simple Network Management Protocol)

  TCP/IP ネットワークにおいて、ネットワークに接続された通信機器をネットワーク経由で監視・制御する。

TCP の機能

TCP は信頼性のある通信をするために以下のような機能を持つ。

- データバッファリング

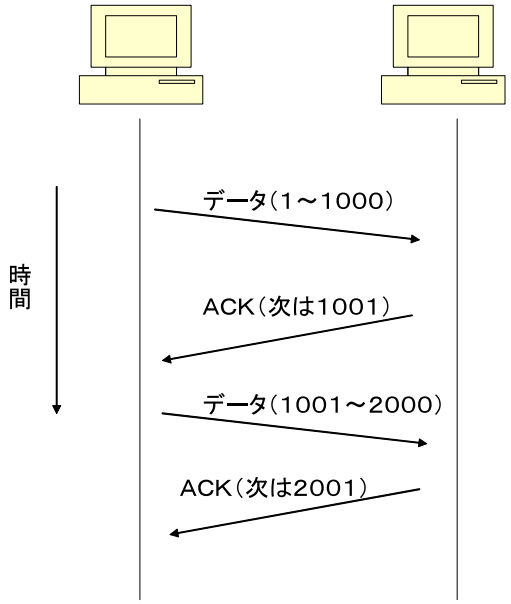

図 2.2: 確認応答（ACK）

TCP では送信したデータが受信ホストに到達したとき、受信ホストは送信ホストにデータが到達したことを知らせる。これを確認応答（ACK）という。
TCP ではデータを送信するときにデータをバッファと呼ばれるメモリ領域にコピーし ACK を受け取るまでの間、保存しておく。このように TCP は送信したデータのうち、受信側のホストに到達したことを確認できていないデータはバッファリングするので、次に説明するデータの再送に備えることができる。

- 失われたデータの再送

  TCP ではデータを送信したホストは、データが相手に届いたかを確認するためにの送信後に確認応答を待つ。確認応答されれば、データは相手のホストに到達したということになる。もし、確認応答されなければ、データが喪失した可能性がある。そこで、ある一定時間以内に確認応答が返ってこない場合には、データが喪失したと判断してもう一度データを送信する。

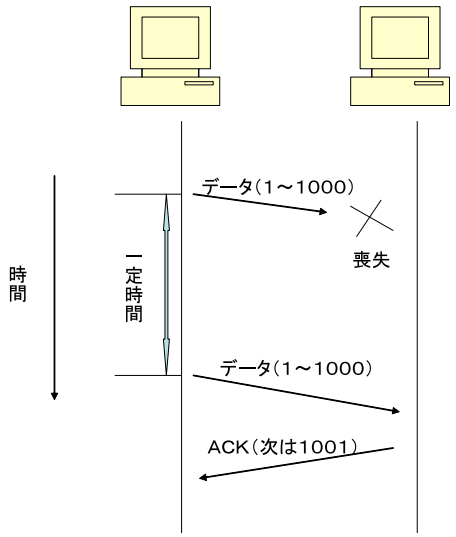

図 2.3: データが喪失した場合

ただし、確認応答されないのはデータが喪失した場合だけとは限らない。確認応答が喪失した場合も確認応答はされない。この間合には、データは届いているがやはりデータを再送することになる。しかし、受信側でデータの重複を制御し、すでに受信したデータが来たら破棄する。

これらの確認応答や再送制御、重複制御などは、すべてシーケンス番号を使って処理が行われる。これは、送信するデータをオクテット単位でカウントして送信時にヘッダに番号を付けて送信する。なお、このシーケンス番号の初期値は 0 から始まるわけではなく、コネクションの確立時に乱数で初期値をきめる。そして、それ以降は、送信データをオクテット単位で数えて、その値にシーケンス番号を加算していく。受信側では、受信したデータのシーケンス番号を調べ、次に自分が受信すべき番号を確認応答として返す。このように TCP ではシーケンス番号を使って信頼性のある通信を実現する。

- コネクション管理

  TCP はコネクション指向の通信を提供する。コネクション指向とは、通信に先立って、通信相手との間に通信を始める準備をしてから通信を行うことである。

  UDP はコネクションレスなので、相手に通信していよいかどうかの確認を求めることなく、いきなり UDP データグラムを送信する。それに対して、TCP は通信前にコネクションの確立要求のパケット

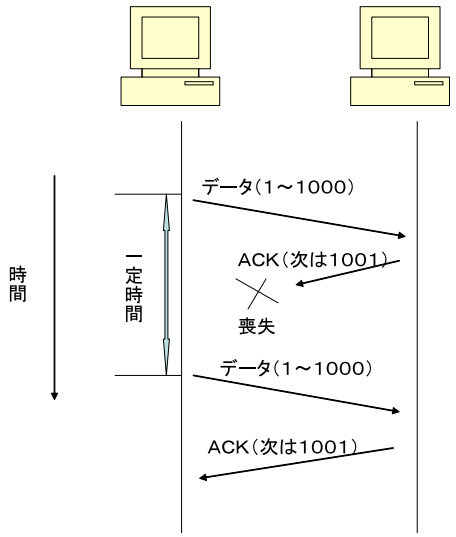

図 2.4: ACK が喪失した場合

を送信してい応答確認を待つ。相手から応答確認が送られてきた場合には通信が可能になるが、確認応答が送られてこなければ通信を開始することはできない。また、通信が終了したときには、コネクションの切断処理を行う。

TCP では、コネクションを管理するために TCP ヘッダの制御用のフィールドを利用する。また、コネクションの確立と切断には最低でも 6 つ異常のパケットがやり取りされることになる。

TCP ソケット

ソケット [?] とは TCP/IP を扱って通信を行うために、アプリケーション層と TCP 層（トランスポート層）の間に定義される API であり、ソケット API と呼ばれることもある。ソケットを使えば、ファイルの読み書きを行うのと同じようにネットワーク経由でデータの送受信を行うことができる。

ソケットの実体は OS(Operating System) のカーネル内にあるメモリ領域である（TCP ソケット）。そこに TCP コネクションを特定するための以下のような情報が保持される。

- ソケットに関連づけられたローカル IP アドレス、リモート IP アドレス、ローカルポート番号、リモートポート番号
- プログラムへの配信を待つ受信データの FIFO キュー（recieve buffer)

と受信ホストに届いたことが確認されていない送信データの FIFO キュー（send buffer）

- TCP ハンドシェイクのプロトコル状態情報

アプリケーションはソケット API を利用して間接的に TCP ソケットを扱うことで、TCP/IP での通信を行うことができる。（図 2.1) Java では

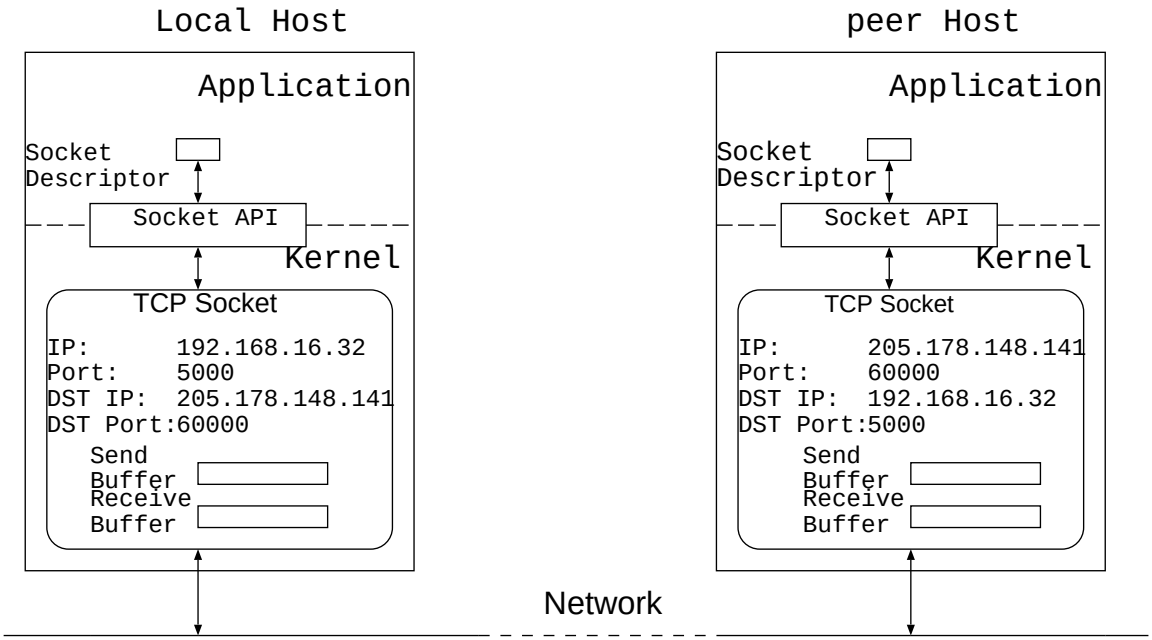

図 2.5: TCP ソケット

TCP による通信を実現するために、Socket と ServerSocket という 2 つのクラスが用意されている。TCP で接続をしている一方の端が、Socket クラスのインスタンスということになる。TCP で接続をしている両端は、それぞれ IP アドレスとポート番号によってお互いを識別することにより抽象的な双方向のチャネルを形成する。

TCP はコネクション指向なので、通信を開始する前にセットアップの手順を経なければならない。クライアント側の TCP がサーバ側の TCP に接続要求を送信することによって通信が開始する。サーバ側は ServerSocket クラスのインスタンスがクライアントからの TCP 接続要求を待ち受け、接続要求を受け付けると新しい Socket クラスのインスタンスを作成する。

接続された Socket クラスの各インスタンスには、InputStream クラスと OutputStream クラスが関連づけられる。ソケットを介してデータを送信するときは OutputStream クラスなどを利用する。データを受信すると

きは InputStream クラスなどを利用する。

アプリケーションが OutputStream クラスに対して書き込みを行うと、TCP ソケットはデータを送信する前に、データを send buffer にコピーする。このデータは通信相手から ACK（acknowledgement) を受け取るまでの間、保存される。一方、TCP ソケットはデータを受信するとデータを receive buffer にコピーして ACK を返す。データはアプリケーションに渡されるまでの間、保存される。つまり、TCP では送信側のホストにおいて、アプリケーションからカーネルに渡されたデータのうち、受信側ホストのアプリケーションに渡されていないデータは送信側の send buffer または受信側の receive buffer のどちらかに保存されていることになる。

receive buffer と send buffer のサイズは java では Socket クラスの getReceiveBufferSize メソッドと getSendBufferSize メソッドによって取得できる。receive buffer が満杯になると、TCP ソケットは通信相手に送信を一度停止するように伝える。アプリケーションにデータが渡されて、receive buffer に空きができたら再び受信を受け付けられるということを相手に知らせる。一方、send buffer が満杯になると、通信相手から ack を受取って send buffer に空きができるまでの間、アプリケーションがそれ以上データを送信できないようブロックする。つまり、アプリケーションはローカルの send bufer とリモートの receive buffer を合わせた大きさ以上のデータを一度に TCP ソケットに渡すことはできない。そのため、送信側のホストでアプリケーションからカーネルに渡されたデータのうち、受信ホスト上のアプリケーションには渡されていないデータは最大でもこの大きさを越えることはない。

TCP コネクションが失われる理由

以下のような理由で TCP コネクションは失われてしまう。一度コネクションが失われると、アプリケーションはそのコネクションに関連する TCP ソケットを使うことはできない。

- 再送回数

  TCP では信頼性のある通信をするために、通信の途中で失われたデータを再送する。再送しても確認応答がとれない場合には、再度再送するが、データの再送を無限に繰り返すことはない。特定の回数再送を繰り返しても確認応答がない場合には、ネットワークや相手のホストに異常が発生していると判断し、強制的にコネクションを切断する。そしてアプリケーションに通信が異常終了したことを通知する。

- so linger オプション

このオプションにゼロでない整数のタイムアウトを指定して有効にすると、close() はブロックされて、ピアに書き込まれる全データの転送、および承認を延期させ、その時点でソケットを閉じる。遅延タイムアウトに到達した時点で、ソケットは TCP RST で強制的に閉じられる。タイムアウトゼロでこのオプションを有効にすると、即座に強制的にソケットを閉じてコネクションを破棄する。

- 通信相手のリセット

  存在しないコネクションや破棄されたコネクションを参照しようとするとき、または、アプリケーションがコネクションを切断したとき、TCP は通信相手にリセットメッセージを送信する。TCP ソケットはリセットメッセージを受け取ると強制的にコネクションを切断する。

- keep alive オプション

  TCP ソケットを介してデータの送受信が行われていないときはコネクションが正常であるかどうかを確認できない。TCP ソケットに KeepAlive オプションを設定すると、ソケットを介してどちらの方向にもデータが 一定時間の間交換されていない場合、TCP は自動的に KeepAlive プローブをピアへ送信する。このプローブは、ピアが応答する必要のある TCP セグメントであり、ピアからの応答がないとコネクションに異常が発生したと判断してコネクションを切断する。

TCP コネクションが失われるケース

モバイルコンピューティングの環境では以下のような事が行われるためにコネクションが失われてしまう。

- ネットワークの物理的な切断

  ケーブルが抜ける、ルーターがダウンするなど、ネットワークが物理的に切断されるとコネクションは強制的に切断される。この原因はデータの再送回数が一定数に達することによる。そのため、アイドル状態のときはネットワークが物理的に切断されていてもコネクションの切断は行われない。

- IP アドレスの変更

  モバイルホストは場所の移動によって、今まで使っていた IP アドレスが変更されることがある。このとき、通信相手がデータを送信

しても、相手に届けることはできないので再送回数が一定数に達してコネクションは強制的に切断される。

- コンピュータの故障

  通信相手のホストがダウンしている間にデータを送信するとデータの再送回数が一定に達してコネクションは強制的に切断される。また、ホストが復帰した後は、リセットメッセージが返えされるため、やはいコネクションは強制的に切断される。

TCP は失われたデータの再送機能を持ち、信頼性のある通信を行うことができるが、以上のようなケースにより TCP コネクションが失われると、TCP ソケットは破棄される。この結果、アプリケーションは再びそのコネクションを使って通信することはできなくなる。

ReliableSocket は、通信中の TCP ソケットの情報（サーバ側のポート番号、再接続専用のポート番号、送受信バッファの内容）を保存し、TCP コネクションが切断されたら待機状態となる。そして、ネットワークに再接続したら、保存してある情報をもとに新しいコネクションを張るので失われた TCP コネクションを回復することができる。その後ネットワーク切断時に失われたデータを再送し、通信が続きから再開できるようにする。

2.3.2　MobileIP

MobileIP とは

モバイル IP[?] は、IP ネットワーク上でホストがネットワーク間を移動した時に、IP アドレスの管理と移動先への通信パケットの転送を自動化する技術である。これを使えば、どのネットワークに接続されていても、あたかも自分のオフィスのネットワークに接続されているかのようにソフトの設定を変えずに端末を利用できる。

通常の TCP/IP 通信では、IP アドレスは所属するネットワークのアドレス体系によって割り振られているため、端末が移動すると IP アドレスが変わり、同一のセッションを維持することはできない。それでも、DHCP を使えば、動的に IP アドレスを割り振られるため、同じパソコンを別なネットワークに移動させても Web ブラウジングや電子メール等を利用することはできる。

しかし、DHCP ではパソコン同士が直接通信するテレビ会議や IP 電話は使えない。というのも新しい IP アドレスに変更されると、変更前のコネクションは無効になり続けて使うことはできないからである。IP アド

レスが変わるたびにコネクションは切断されるので続けて通信を行うことはできない。

モバイル IP はこの問題を解決するために提案された技術である。すなわち、通信相手の端末が異なる場所、異なるネットワークに移動しても、ホームエージェントと呼ばれる位置管理機能を備えたルータ経由で、常にパケットが届くようにするものである。モバイル IP を使えば、移動しながら通信を続けることができる。さらに通信しながら無線 LAN から携帯電話ネットワークへと異なる通信回線に切り替えることもできる 。

モバイル IP の構成要素

- モバイルホスト (MH)

  ネットワークに応じて接続点を変更するホスト。

- ホームエージェント（HA）

  モバイルホスト向けのデータグラムを取得して気付アドレス（移動先アドレス）に転送する、モバイルホストのホームネットワーク上のノード（ルータまたはサーバ）。ホームエージェントは、モバイルホストの現在の場所情報も保持している。

- フォーリンエージェント（FA）

  モバイルホストが登録されている間、そのモバイルホストに経路指定サービスを提供する、モバイルホストの移動先ネットワーク上にあるノード。

モバイル IP のメカニズム

モバイル IP では、本来のネットワークをホーム・ネットワーク、移動先で一時的に使うネットワークをフォーリン・ネットワークと呼び、移動先の位置情報をホームエージェントに登録する。

モバイルホストは IP コネクティビティを取得すると最初に現在自分がいるネットワークが、ホーム・ネットワークか、それともフォーリン・ネットワークであるかを識別する必要がある。

この目的に利用するメッセージが Agent Advertisement メッセージである。Mobile Agent Advertisement メッセージは、ICMP ルータ発見メッセージを拡張したものである。フォーリン・エージェントはこのメッセージを定期的に送出する。何らかの理由により移動端末が Mobile Agent Advertisement メッセージを受け取れない場合、自らルータに対して Mobile Agent Advertisement メッセージの送信をリクエストすることもできる。

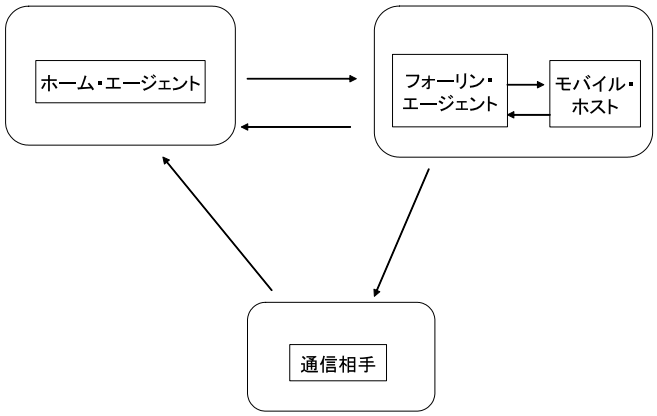

図 2.6: TCP ソケット

その際に利用するメッセージが Mobile Agent Solicitation メッセージだ。このメッセージを受け取ったフォーリン・エージェントは速やかに Mobile Agent Advertisement メッセージを送出することになる。

モバイルホストはこうやって受信した Mobile Agent Advertisement メッセージの内容から、自分がホーム・ネットワークにいるのか、それともフォーリン・ネットワークにいるのか判断することができる。

自分がフォーリン・ネットワークにいることを検出したら、次に自分の現在位置をホーム・エージェントに登録するため、Registration Request メッセージを送出する。このメッセージには UDP のポート 434 番を使用する。Registration Request メッセージはフォーリン・エージェントを介して（場合によっては直接ホーム・エージェントに送信されることもある）ホーム・エージェントに送信される。

ホーム・エージェントは、こうして受け取った Registration Request メッセージから当該ホストがいるネットワークを検出し、自らが管理する転送先テーブルに登録する。この処理が終わると、ホーム・エージェントは位置登録の結果を Registration Response メッセージに書き込み、フォーリン・エージェントに送り返す。フォーリン・エージェントはそのメッセージからパケットの受信準備を行った後、モバイルホストに Registration Response メッセージを送り返す。

以上のメッセージ交換によって、ホーム・エージェントにはモバイルホストあてのパケットの転送先が記録され、またフォーリン・エージェントはホーム・エージェントから受け取ったパケットの送出先が記録される。これらの情報があることによって、初めて移動端末のホーム・アドレスあてに送出した IP パケットが、移動先に転送することが可能となる。

なお、モバイルホストが通信相手に送出するパケットは、ホーム・エー

ジェント〜フォーリン・エージェント間のトンネルは通らない。通常の IP パケットの配送ルートによって配送させる。また、このパケットの送信元アドレスにはホーム・アドレスが記されている。このことは、通信相手が端末の移動の有無や移動先をまったく感知しなくてよいことを示している。

モバイル IP の問題点

- ネットワークプロトコルの変更を要する

  モバイル IP を実現するためには HA、FA といった固定ノードを設置し、MH 及びそれらの端末の OS にはモバイル IP の機能を実装するように変更を加える必要がある。追加するソフト（HA、FA）やカーネルレベルでの変更を必要とするために技術導入によって影響の及ぶ範囲が大きくなってしまう。

- TCP コネクションのエラーを検出できない

  モバイル IP は TCP の下位に位置する IP 層において IP アドレスの変更を TCP ソケットから隠蔽する。そのため、TCP コネクションのエラーを検出することはできない。また、IP アドレスが変更するときにデータが失われる場合、データの回復をすることができない。

2.3.3　ReliableSockets

本研究の ReliableSocket は ReliableSockets[**?**, **?**] を参考にして実装した。特に I/O 処理と切断時に失われたデータの回復方法は ReliableSockets のアルゴリズムを参考にした。本研究における ReliableSocket との違いは次のようになっている。

- 通信相手が ReliableSockets を使っているかの確認

  ReliableSockets は通信をはじめるときに相手が ReliableSockets を使っているかを調べる。相手が通常のソケットを使っている場合は ReliableSocket の機能を使わないで通信を開始する。ReliableSocket では互いに ReliableSocket を使用していると仮定して通信をはじめる。

- 制御用コネクション

  ReliableSockets では制御用コネクションで互いにプローブを交換し、数プローブ続けて受信することができないと障害が発生したとみなして待機状態にする。ReliableSocket は制御用コネクションは張ら

ず、コネクションの障害は I/O 処理中に発生する例外によって検出する。また、制御情報（再接続専用のポート番号の値、read カウンタの値など）はデータ用コネクションで交換する。制御用コネクションを張らないために余分なリソースは使わずにすむという利点がある。しかし、コネクションの障害を I/O 処理中に検出するために、障害がおきたときに I/O 処理を行っていないような場合は I/O 処理が行われるまで障害の検出が遅れてしまう。ReliableSockets でも NAT やファイアーウォールなどにより、制御用コネクションが張れない場合はデータコネクションで切断を検出する。

- 再接続処理

  ReliableSockets はネットワークコネクションの障害によって待機状態になると互いに今まで通信してい相手（ソケット）のアドレス（IP アドレスとポート番号）に繰り返し再接続する。また、相手からの再接続要求を受け付けるために今まで使用していた IP アドレスとポート番号にサーバソケットをバインドする。このため、どちらか一方のホストの IP アドレスが変更されたとしてもコネクションを張ることができる。しかし、この方法だと両ホストの IP アドレスがどちらも変更していない場合はコネクションが 2 本張られることになるので、その場合はサーバ側の ReliableSocket がどちらか一方を選んで他方をクローズする。本研究では実験によってネットワーク切断時の例外はネットワークを切断するホストでしか検出することができなかった。そのため、例外が発生したら互いに再接続するという実装にはなっていない。本研究ではクライアント/サーバ型の通信を仮定し、モバイルホストはクライアント側だけを想定している。つまり、ネットワークを切断して例外を検出し、再接続するのはクライアント側である。サーバの IP アドレスが変更されない限り、クライアント側の IP アドレスは変更されても再接続して通信を再開することができる。

第3章　設計と実装

3.1　システムの概要

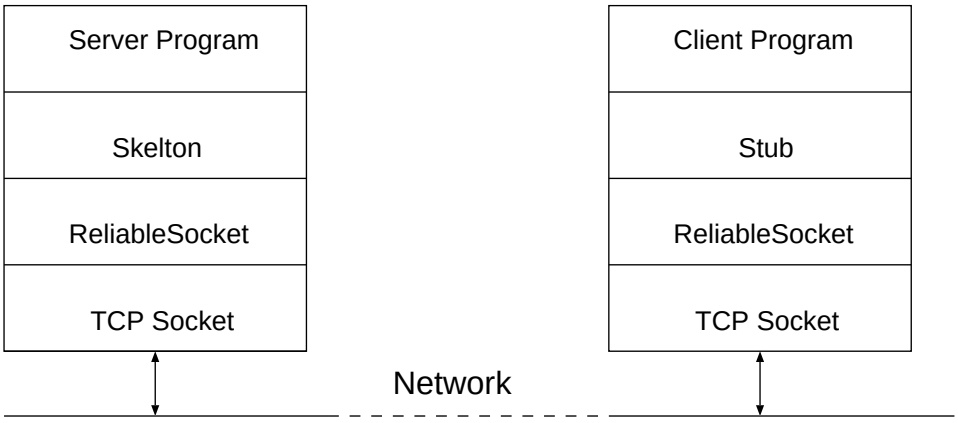

図 3.1: システムの構成

Java 分散ミドルウエアを高信頼化するためにソケットを拡張して高信頼化を行った (ReliableSocket)。ReliableSocket は java.net.Socket に以下の機能を追加したものである。

- 通信中の TCP コネクションの情報（サーバの IP アドレスと再接続専用のポート番号、送受信バッファの内容）を保存する
- コネクションが失われたら自動的にサーバに再接続する。そしてネットワークに再接続したら、保存している情報をもとに新しいコネクションを張る
- 再接続してきた相手が今まで通信をしていた相手であるかを確かめるために認証を行う
- 切断時に失われたデータを回復し、通信が続きから再開できるようにする

分散ミドルウエア内で使用するソケットを ReliableSocket と置き換えることで分散ミドルウエアを高信頼化することができる。本研究では実際に JavaRMI に ReliableSocket を組み込んで高信頼化を行った。ReliableSocket を JavaRMI に組み込むためには、使用するソケットとして ReliableSocket を返すようにカスタム RMISocketFactory を実装する。そして、サーバアプリケーションにおいて、カスタム RMISocketFactory を使用するリモートオブジェクトを作成してエクスポートする。図 3.1 にシステムの構成を示す。

3.2　ReliableSocket

3.2.1　ReliableSocket の概要

ReliableSocket は Java.net.Socket を継承しているので、アプリケーションは通常の Java.net.SocketAPI と同様にして ReliableSocket を使用することができる。図 3.2 に ReliableSocket の構成を示す。

ReliableSocket は Reliable I/O Stream を使って I/O 処理を行う。ReliableOutputStream はデータ保存用バッファと送信したバイト数を数える write カウンタを持つ。ReliableInputStream は受信したバイト数を数える read カウンタを持つ。

ReliableOutputStream はアプリケーションからデータを渡されると、データを送信する際に、データをバッファに保存する。そして送信したバイト数分だけ write カウンタを増やす。ReliableInputStream はデータを受信すると受信したバイト数分だけ read カウンタを増やす。データ保存用バッファは満杯になると古いデータを捨て、新しいデータを入れるためのスペースを作るが、相手がまだ受信していないデータはすべて保存しておけるだけの十分な大きさを持っている。

ネットワーク切断時の例外の検出と再接続、そして再接続の受け付けは Reliable I/O Stream の read メソッドと write メソッドで行う。read メソッドと write メソッドは java.nio.Channels の SocketChannel を使ってノンブロッキングモードで送受信を行う。メソッド内では、送受信を行う SokcetChannel と再接続を受け付ける ServerSocketChannel をセレクタに登録し、I/O 処理の最中でも相手からの再接続の要求を受け付けられるようにする。。ServerSocketChannel は ReliableSocket クラスで生成され、Reliable I/O Stream に渡される。

Reliable I/O Stream はネットワーク切断による例外を検出すると待機状態となり自動的に通信相手の write メソッドまたは read メソッド内にバインドしているサーバーソケットに再接続する。再接続することができ

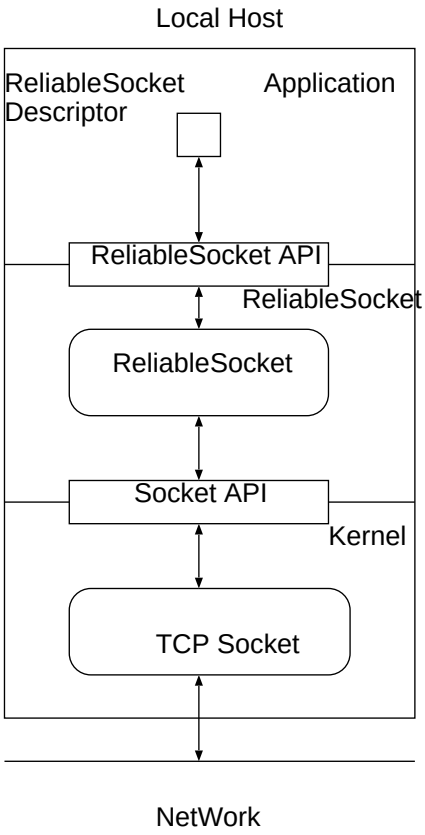

図 3.2: ReliableSocket の構成

たら、write カウンタと read カウンタの差を調べることにより切断時に失われたデータを回復して通信を続きから再開する。

3.2.2　ReliableSocket の構成

- ReliableSocket

  Java.net.Socket を継承する。I/O 処理は Reliable I/O Stream で行う。

- ReliableServerSocket

  Java.net.ServerSocket を継承する。accept メソッドで ReliableSocket 型のオブジェクトを返す。

- ReliableInputStream

  Java.io.FilterInputStream を継承する。read メソッドをオーバーライドすることによりリライアブルな I/O 処理を行う。Java.net.Channels の SocketChannel でノンブロッキングモードで受信を行う。

- ReliableOutputStream

  Java.io.FilterOutputStream を継承する。write メソッドをオーバーライドすることによりリライアブルな I/O 処理を行う。Java.net.Channels の SocketChannel でノンブロッキングモードで送信を行う。

3.2.3　ReliableSocket の使用法

ReliableSocket を利用したプログラムの例として HelloWorld!を送受信するクライアント・サーバプログラムを示す。

サーバプログラム

以下のように ServerSocket オブジェクトの代わりに ReliableServerSocket オブジェクトを生成するところ以外は java.net.socket と同様である。

```
import java.net.*;
import java.io.*;
public class HelloServer{
 ServerSocket ssock;

 public HelloServer() throws IOException{
```

```
    ssock = new ReliableServerSocket(9000);
 }

 public void run(){
    try{
       Socket sock = ssock.accept();
       ObjectInputStream in = new ObjectInputStream(sock.getInputStream());
       String msg = (String)in.readObject();
       System.err.println(msg);
       ObjectOutputStream out = new ObjectOutputStream(sock.getOutputStream());
       out.writeObject("HelloWorld!");
       out.flush();
     }
     catch(Exception e){
     }
  }

 public static void main(String args[]) throws IOException{
     new HelloServer().run();
 }
}
```

クライアントプログラム

以下のように Socket オブジェクトの代わりに ReliableSocket オブジェクトを生成するところ以外は java.net.Socket と同様である。

```
import java.net.*;
import java.io.*;
public class HelloClient{
  Socket sock;

  public HelloClient() throws IOException{
     sock = new ReliableSocket("localhost", 9000);
  }

  public void run() throws IOException{
     try{
       ObjectOutputStream out = new ObjectOutputStream(sock.getOutputStream());
```

```
        out.writeObject("HelloWorld!");
        out.flush();
        ObjectInputStream in = new ObjectInputStream(sock.getInputStream());
        String msg = (String)in.readObject();
        System.err.println(msg);
      }
      catch(Exception e){
      }
  }

  public static void main(String args[]) throws IOException{
      new HelloClient().run();
  }
}
```

3.2.4　ReliableSocket の機能

コネクションの障害の検出

コネクションの障害は I/O 処理中に発生する例外によって検出される。本研究では、クライアント/サーバ型の通信を想定し、クライアント側が移動しサーバ側は移動しないという状況を想定ている。実験によって、ネットワーク切断時の例外は、移動によってネットワークを切断するクライアント側でしか発生しないことがわかった。サーバ側はこのとき処理を停止し、クライアントからの応答待ちの状態となっている。クライアント側の Reliable I/O Stream は例外を検出したら再びコネクションを張れるまでサーバ側の Reliale I/O Stream に再接続する。Exception を利用することによって処理を待機状態にし、再接続するアルゴリズムを write メソッドを例にして以下に示す。

```
public void write(byte[] b , int off, int len) throws IOException{
   while(true){
      try{
          write、accept //データの送信と後述する再接続要求の受
け付け
          break //送信できたらループを抜け終了
      }
      catch(IOException e){
        while(true){
          try{
```

```
            sleep//しばらく時間をおく
            connect //再接続
            break   //コネクトできたらループを抜ける。
          }
          catch(SocketException ex){
        }
      }
    }
  }
}
```

以上のような処理を行うことによって、ネットワーク切断の例外が発生すると内側のループに入りしばらく時間をおいて再接続をする。このときネットワークが回復していなければ再接続することができず再び例外が発生するのでループを抜けることができない。ネットワークが回復し、コネクションを張ることができたら内側のループを抜け、先ほど送信できなかったデータを送信してメソッドは終了する。

再接続

先に説明したようにクライアント側は例外を検出するとサーバ側に再接続する。再接続要求の受け付けは ServerSocketChannel をノンブロッキングモードにして I/O 処理と平行して行われる。

ReliableSocket は再接続要求を受け付けるためのサーバソケットを用意し、適当なポート番号ににバインドする。ポート番号は各 ReliableSocket オブジェクトごとに一意であるため、ポート番号によってコネクションが識別される。ポート番号は通信を始める前交換されて、切断による例外を検出したらポート番号と今まで通信していた相手の IP アドレスをもとに再接続する。この方法により、サーバ側の IP アドレスが変更しない限り再接続して新しいコネクションを張ることができる。

失われたデータの回復

ネットワーク切断時には、両ホストにおける TCP ソケットの送信バッファと受信バッファにあるデータが失われてしまう。

ReliableSocket は recover メソッドで切断時に失われたデータを回復する。recover メソッドでは相手の受信カウンタの値を受取り、送信カウンタの値と比較して差があればデータを再送する。このようにすることで、切断時に失われたデータを回復することができる。また、データの順序が

狂わないように、クライアント側の Reliable I/O Stream は例外を検出して再接続できたらすぐに recove メソッドを呼び出す。サーバ側の Reliable I/O Stream は再接続を受け付けたらすぐに recover メソッドを呼び出す。

切断時に失われるデータは、TCP ソケット内の送信バッファと受信バッファに残っているデータである。これはつまり、データを送信する側のホスト上においてアプリケーションからカーネルに渡されたデータのうち、受信側ホスト上のアプリケーションには渡っていないデータということになる。2.3.1 で説明したように、このようなデータの大きさは送信側のホストにおける送信バッファサイズと、受信側のホストにおける受信バッファサイズを合わせた大きさを越えることはない。このため、切断時に失われるデータを回復するためには、この大きさのバッファにデータを保存しておけばよい。

ReliableSocket は始めに、お互いの TCP 受信バッファサイズを交換する。その後 ReliableSocket はローカルの TCP 送信バッファサイズと受け取った相手の TCP 受信バッファサイズを合わせた大きさのデータ保存用バッファを作る。

保存用バッファは ReliableOutputStream に渡される。アプリケーションが ReliableOutputStream の write メソッドを呼び出すと、ReliableOutputStream は送信する際に、アプリケーションから渡されたデータを保存用バッファに保存する。

3.2.5　new IO

read ブロック

java.net.Socket による通信では、read メソッド、accept メソッド、write メソッドで通信ができるまでブロックしてしまう。

accept メソッドはクライアントがコネクトするまで待ち状態となる。同様に read メソッドはデータが受信できるまで待ち状態のままとなる。write メソッドは送信バッファに送信するデータをすべて入れられるまで待ち状態となる。

このような状態をブロッキング状態という。前述したようにサーバ側ではネットワーク切断時の例外を検出することはなく、相手から切断されたということを感知することができない。そのため、クライアントがデータを送信し、サーバがデータを受信している最中に、クライアントがネットワークを切断するとサーバは read メソッドでブロックしてしまう。この結果、クライアントが再接続しようとしても、accept メソッドに制御が移らないので再接続は受け付けられない。accept の処理を別スレッドで行ったとしても read メソッドは永遠にブロッキング状態から抜け出すことが

できず、処理を再開させることができない。

以上のような問題を解決するために、ReliableSocket は java.nio.channels の SocketChannel を用いて通信を行う。new IO クラスは java2SE, v1.4.0 から導入され、ノンブロッキング通信を可能にるために SocketChannel や selector を提供する。

ノンブロッキングもードで channel を使用することにより、再接続要求の受け付けを I/O 処理と平行して行うことができる。このため、再接続専用のスレッドが必要ないので余分なスレッドを使わなくてすむ。

ノンブロッキングモードでは accept メソッド、read メソッド、write メソッドのいずれもブロックしない。ノンブロッキングモードにおける 3 つのメソッド振る舞いを示す。

- accept

  クライアントからのコネクト要求がなければそのまま抜けてしまう。

- read

  データが来なければそのまま抜けてしまう。

- write

  送信バッファに入れられる分のデータを入れたら終了する。そのため、SocketChannel で write するとデータがまったく書き込まれない場合もある。

接続要求やデータの有無、書き込み可能かどうかといったことを監視するために Java.nio.Slector クラスの selecto メソッドを使用する。Selector に SocketChannel や ServerSocketChannel を登録することで、接続要求やデータの有無を判断し、適宜 read, write, accept メソッドを呼ぶ。

ノンブロッキングシステム

図 3.3 にノンブロッキングソケットを使ったシステムの構成を示す。

- Server リクエストを受けるアプリケーション
- Client サーバにリクエストを出すアプリケーション
- SocketChannel

  サーバとクライアントの間の通信経路。サーバの IP アドレスとポート番号によって決められる。データはこの経路を buffer として移動する。

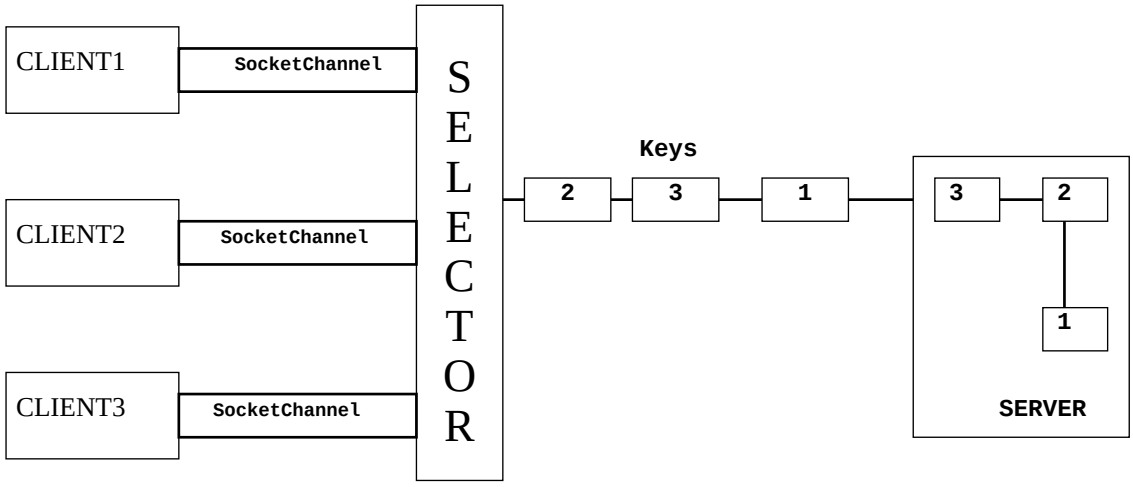

図 3.3: ノンブロッキングソケットのアーキテクチャ

- Selector

  登録された ServerSocketChaneel と SocketChannel を全て監視し、いずれかの SocketChannel でクライアントから要求があるまで待つ。クライアントから要求が来たら、その要求の種類（クライアントが接続してきたか、データの送信または受信を行うか）を調べ、結果を key として返す。サーバはこの key をもとに要求された作業を行う。

- key

  クライアントからの要求の種類とクライアントの情報をもつ。図の各 key の番号はクライアントの番号に対応している。

クライアントからの要求の種類は以下の 4 つである。

- connect
- accept
- read
- write

プログラム例

以上のノンブロッキングシステムのプログラム例を示す。

```
//サーバの無限ループ
for(;;) {
  // イベントを待つ
  selector.select();
  // key を得る
  Set keys = selector.selectedKeys();
  Iterator i = keys.iterator();

  // それぞれの keys に対して
  while(i.hasNext()) {
    SelectionKey key = (SelectionKey) i.next();

    // 現在の key を取り除く
    i.remove();

    //isAccetable = true のとき
    //接続を受け付ける
    if (key.isAcceptable()) {
      //クライアントの socket channel を得る
      SocketChannel client = server.accept();
      // ノンブロッキングモードに設定
      client.configureBlocking(false);
      //セレクタに登録
      // 新しく得たクライアントからの read と write を監視する
      client.register(selector, SelectionKey.OP_READ || SelectionKey.OP_WRITE);
      continue;
    }

    // isReadable = true のとき
    // データの受信
    if (key.isReadable()) {
      SocketChannel client = (SocketChannel) key.channel();
      client.read(buffer);
      continue;
     }
    //isWritable=true のとき
    //データの送信
    if(key.isWritable()){
```

```
        SocketChannel client =(SocketChanel)key.channel();
        client.write(buffer);
        continue;
    }
```

write メソッドのアルゴリズム

ReliableOutputStream の write メソッドではノンブロッキングモードで送信する。そのため、送信バッファの状態によっては、アプリケーションから渡されたデータを全て送信しないで write メソッドが終了してしまうことが起る。

また、送信バッファを越える大きさのデータは送信バッファサイズに分けて送信するようにする。

write メソッドのアルゴリズムを示す。

```
public void  write(byte[] b, int off, int len) throws IOException{
    int off2 = off;
    int num;
    //送信するデータが送信バッファサイズを越えている場合
    if(len > sendBufferSize)
      データを sendBufferSize ごとに分割して write1 メソッドを呼ぶ
      残りが sendBufferSize より小さくなれば break

    //全てのデータが書き込まれるまでループする
     while(true){
       num = write1(b, off2, len);
       if(num == len) break;
       off2 += num;
       len -= num;
    }
 }

private int write1(byte[] b, int off, int len){
    SocketChannel と ServerSocketChannel をレジスタに登録
    while(true){
    escape:
        try{
           selector.select();
           Set keys = selector.selectedKeys();
```

```
            Iterator i = keys.iterator();
            while(i.hasNext()){
                SelectionKey key = (SelectionKey)i.next();
                i.remove();
                //送信
                if(key.isWritable()){
                   num = sc.write(ByteBuffer.wrap(b, off, len));
                   send_length += num; //送信カウンタを増やす
                   データのコピー
                 }
                 //再接続を受け付ける
                 else if(key.isAcceptable()){
                    sc = ssc.accept();
                    新しいソケットチャンネルをセットする
                    データの回復
                    break escape;
                  }
              }
              return num;
            }
            catch(IOException e){
                while(true){
                     try{
                         .
                         .
                      }
                      catch(Exception e){

                      }
                 }
            }
        }
}
```

3.3　JavaRMI における ReliableSocket の使用

3.3.1　JavaRMI

JavaRMI とは

RMI（Remote Method Invocation) は他のマシンにある Java で記述されたメソッド（リモートメソッド）をネットワークを介して呼び出す仕組みである。RMI のような分散オブジェクト技術を使用することによりネットワークをまたがる分散アプリケーションの構築が可能になる。JavaRMI には以下のような特徴がある。

- 変更に対する柔軟性

  RMI ではサーバとクライアントは Java のインターフェースで結合される。これはインターフェースが変更されない限り、サーバの処理に変更があってもクライアントを変更する必要がなく、また、サーバの変更なしでクライアントを変更できることを意味する。したがって RMI では、このようにサーバ、クライアントの変更に対して柔軟なシステムを構築できる。

- 記述の容易性

  RMI を用いてサーバ、クライアントプログラムを記述することは簡単である。サーバプログラムでは、それが RMI のサーバであることを宣言するための数行のコードを記述するだけである。クライアントプログラムではサーバプログラム、すなわちサーバ、オブジェクト、リモートオブジェクトの参照として得られるが、これは通常のオブジェクトと何らかわりなく扱うことができる。

- 可搬性

  RMI は JDK1.1 以降でコアパッケージとして提供されている。したがって、RMI を使用して作成されたプログラムは JDK1.1 以降の環境があれば、他のどのようなシステムにおいても動作させることが出来る。

- データ送受信の容易性

  RMI ではメソッドを呼び出す際の引数、戻り地として、オブジェクトを受け渡す子とができる。したがってハッシュテーブルのような複雑なデータ構造でも、ただ 1 つの引数として取り扱うことができる。例えば、ソケットを使用してデータを転送する場合と比較して、データの分解、再構成といった、余分なコードは一切不要となる。

- 並列処理

  RMI はマルチスレッド対応であり、複数のクライアントの要求は並列に処理される。

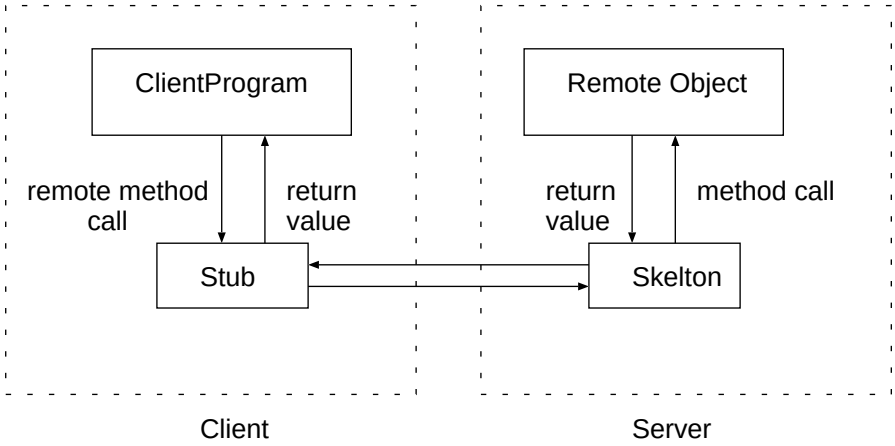

図 3.4: RMI の仕組み

RMI の用語の説明

- リモート・インターフェース

  Remote インターフェースを直接または間接的に継承したインターフェース。他の Java 仮想マシンから呼び出されるメソッドはここに定義する必要がある。

- リモートオブジェクト

  リモート・インターフェースを実装したクラスのオブジェクトで、かつ他の Java 仮想マシンからそのリモート・インターフェースが参照可能なもの。

- リモート・メソッド

  リモート・オブジェクトに実装されたメソッドで、かつリモート・インターフェースに定義されているもの。

- ローカルオブジェクト

  参照元のオブジェクトと同一の Java 仮想マシン上にあるオブジェクト。

- レジストリ

  名前付けされたリモート・オブジェクトの登録、検索を行うネーミングサーバ。サーバはリモート・オブジェクトに名前をつけてレジストリに登録し、クライアントはレジストリに対して名前でリモート・オブジェクトを検索し、その参照を得る。

RMI の具体的な使用例

JavaRMI で HelloWorld!を出力するプログラム例を示す。

リモートインターフェース

- public である。
- Remote インターフェースを継承する
- RemoteException 例外を送出する

モートインターフェース（Hello.java)

```
public interface Hello extends java.rmi.Remote {
    String sayHello() throws java.rmi.RemoteException;
}
```

サーバプログラム

リモートインターフェース（Hello.java) を実装するクラス

```
import java.rmi.*;
import java.net.*;

public class HelloImpl extends UnicastRemoteObject
     implements Hello{

     public HelloImpl() throws RemoteException {
          super();
     }

     public String sayHello() throws RemoteException{
           return "Hello World!";
     }
```

```
}
```

レジストリートへの登録

セキュリティマネジャの設定とサーバクラスのインスタンス化と登録を行う。

オブジェクトエクスポートクラス（Launcher.java）

```
import java.rmi.*;
import java.net.*;

public class Launcher{
    public static void main(String[] args){
         //セキュリティマネジャの作成と設定
         System.setSecurityManager(new RMISecurityManager());
         try{
            //サーバオブジェクト作成
            Helloimpl obj =new HelloImpl();
            //リモートオブジェクトの登録
            Naming.bind("rmi://localhost/HelloImpl", obj);
          }
          catch(Exception e){
            e.pintStackTrace();
          }
      }
}
```

クライアントプログラム

クライアントプログラムはリモートオブジェクトの取得とリモートメソッドの呼び出しを行う。

クライアントプログラム（HelloClient.java）

```
import java.rmi.*;
import java.net.*;

public class HelloClient{
    public static void main(String[] args){
```

```
        Hello remoteObject;
        try{
          remoteObject=(Hello)Naming.lookup("rmi://localhost/HelloImpl");
        }
        catch(Exception e){
           e.printStackTrace();
         }
         try{
           String message = remoteObject.sayHello();
         }
         catch(Exception e){
           e.printStackTrace();
         }
    }
}
```

コンパイルと実行

1．ソースコードをコンパイルする。
2．スタブとスケルトンを生成する（rmic HelloImpl）
3．レジストリプログラムの実行（rmiregistry）
4．サーバプログラムの実行（Java Launcher）
5．クライアントプログラムの実行（java HelloClient）

3.3.2　カスタム RMI ソケットファクトリ

JavaRMI はデフォルトでは Java.net.Socket を使っているが、カスタム RMI ソケットファクトリを実装することによって ReliableSocket を使用するように変更できる。クライアント側、サーバ側でそれぞれのソケットファクトリを作成する必要がある。

クライアント側カスタム RMI ソケットファクトリ: ReliableClientSocketFactory の実装

ReliableClientSocketFactory は、java.rmi.server.RMIClientSocketFactory インタフェースを実装し、ReliableSocket を返す createSocket メソッドを実装する。

クライアントソケットファクトリは、java.io.Serializable を実装してイ

ンスタンスがクライアントに直列化されるようにする必要がある。

プログラム例

```
import java.io.*;
import java.net.*;
import java.rmi.server.*;

public class ReliableClientSocketFactory
    implements RMIClientSocketFactory, Serializable {

    public Socket createSocket(String host, int port) throws IOException {
        return new ReliableSocket(host, port);
     }
}
```

サーバ側カスタム RMI ソケットファクトリ：
ReliableServerSocketFactory の実装

ReliableServerSocketFactory は java.rmi.server.RMIServerSocketFactory インタフェースを実装し、ReliableServerSocket を返す createServerSocket メソッドを実装する。サーバソケットファクトリのインスタンスはクライアントに直列化されないので、サーバソケットファクトリは Serializable インタフェースを実装する必要はない。

プログラム例

```
import java.io.*;
import java.net.*;
import java.rmi.server.*;

public class ReliableServerSocketFactory
    implements RMIServerSocketFactory, Serializable {

    public ServerSocket createServerSocket(int port) throws IOException {
        ReliableServerSocket rss = new ReliableServerSocket();
        rss.bind(new InetSocketAddress(port));
        return rss;
    }
```

```
}
```

アプリケーションでの ReliableSocketFactory の使用

リモートオブジェクトで ReliableSocketFactory を使用するのに必要な追加のステップは次の 2 つだけである。

1. サーバアプリケーションで `ReliableSocketFactory` を使用するリモートオブジェクトを作成してエクスポートする。リモートオブジェクトのスタブへの参照を `RMI` レジストリに保存してクライアントがそれを検索できるようにする 。

2. リモートオブジェクトのスタブを検索してリモートメソッドを呼び出すクライアントアプリケーションを作成する。`ReliableSocketFactory` は、クライアントアプリケーション内で参照される必要はない。クライアントがリモートオブジェクトのスタブを検索すると、`ReliableClientSocketFactory` がクライアントにダウンロードされる

以下では ReliableSocket を使用してクライアントがサーバの HelloWorld! を出力するメソッドを呼び出すアプリケーションを例にする。

ステップ 1：サーバアプリケーションを作成する

このアプリケーションは、次の Hello リモートインタフェースを使用します。

```
public interface Hello extends java.rmi.Remote {
    String sayHello() throws java.rmi.RemoteException;
}
```

このサーバアプリケーションは、Hello リモートインタフェースを実装するリモートオブジェクトを作成してエクスポートし、カスタムソケットファクトリを引数にとる java.rmi.server.UnicastRemoteObject.exportObject メソッドを使ってソケットファクトリを使用します。次に、ローカルレジストリを作成し、そのレジストリ内で、リモートオブジェクトのスタブへの参照を「Hello」という名前でバインドします。

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
```

```
public class HelloImpl implements Hello {

    public HelloImpl() {}

    public String sayHello() {
        return "Hello World!";
   }

    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }

        try {
            /*
             * Create remote object and export it to use
             * custom socket factories.
             */
            HelloImpl obj = new HelloImpl();
            RMIClientSocketFactory csf = new ReliableClientSocketFactory();
            RMIServerSocketFactory ssf = new ReliableServerSocketFactory();
            Hello stub =
                (Hello) UnicastRemoteObject.exportObject(obj, 8000, csf, ssf);

            LocateRegistry.createRegistry(2002);
            Registry registry = LocateRegistry.getRegistry(2002);
            registry.rebind("Hello", stub);
            System.out.println("HelloImpl bound in registry");

          } catch (Exception e) {
             System.out.println("HelloImpl exception: " + e.getMessage());
             e.printStackTrace();
          }
      }
}
```

ステップ 2：クライアントアプリケーションを作成する

クライアントアプリケーションは、サーバアプリケーションが使用するレジストリへの参照を取得します。次に、リモートオブジェクトのスタブを検索して、リモートメソッド sayHello を呼び出します。

```
import java.rmi.*;
import java.rmi.registry.*;

public class HelloClient {
     public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
           System.setSecurityManager(new SecurityManager());
        }
        try {
            Registry registry = LocateRegistry.getRegistry(2002);
            Hello obj = (Hello) registry.lookup("Hello");
            System.out.println(obj.sayHello());
        } catch (Exception e) {
            e.printStackTrace();
        }
   }
}
```

アプリケーションのコンパイルと実行

1. リモートインターフェース、クライアント、およびサーバの各クラスをコンパイルする
2. 実装クラス上で rmic を実行する
3. サーバを起動する
4. クライアントを実行する

3.4　認証

ReliableSocket を使用したシステムでは、ネットワーク切断後も再接続して続きから分散アプリケーションを使用できる。

このとき、再接続してきたホストが以前通信していたホストと同一であることを認証する必要がある。

簡単な認証の方法としては、各 ReliableSocket オブジェクトが自分を特

定するための重複しない key を保持し、通信を始める前に互いの key を交換する。そして、切断後にも同じ key をもっているかどうか確かめることで認証することができる。

key はネットワークを通じてやりとりされるので注意が必要である。インターネットでは、自分のホストから相手のホストまでの間に複数のルータを経由し、ルータ内を通過するデータを読み出すことはそれほど難しくない。また、ルータを通過するデータを故意に改ざんすることもできる。このため key は暗号化してやり取りする必要がある。

3.4.1　暗号化技術

暗号化とは、あるアルゴリズムを使用してデータを加工し、データが第三者には読み取れないようにすることである。データを解読することを複合という。暗号化と複合化には鍵を用いる。代表的な暗号化技術は秘密鍵公開方式と公開鍵暗号方式である。(1) 秘密錠暗号方式
共通錠暗号方式、慣用錠暗号方式ともいう。送信者、受信者の双方が同じ錠を持ち、暗号化、復号化を行う方式である。

特徴

- 暗号化・複合化を高速に行える

  アルゴリズムは文字の置き換え（換字）と文字の入れ替え（転置）を複雑に組み合わせたものである。このような単純なアルゴリズムのため、暗号化・複合化の処理を高速に行うことができる。代表的な方法に DES がある。

- 鍵の管理が難しい

  n 人の送信者が 1 人の受信者と通信を行う場合、受信者は n 個の鍵をすべて持っていなければならない。

- 受信者に秘密鍵を通知しなければならない

  送信者は受信者に鍵を送信しないといけないが、鍵が盗まれてしまったら暗号化をする意味がなくなってしまう。

(2) 公開錠暗号方式
送信者、受信者がそれぞれ別の錠を使う方式である。送信者は受信者の公開錠を使って暗号化し、受信者は受信者の秘密錠を使って複合する。

特徴

- 鍵の配送が不要

公開鍵暗号方式では公開されている受信者の鍵を入手すればよい。鍵の配送が不要なので秘密鍵暗号方式に比べてセキュリティが高くなる。

- 鍵の管理が楽

  受信者は自分の秘密鍵だけを持てばよい。

- 電子署名が実現できる

  送信者がディジタル署名を秘密錠で作成し、データに添付して送信する。受信者は公開錠で復号し、ディジタル署名とデータが同一であるか確認する。ディジタル署名は第三者や受信者には偽造できないようになっており、送信者本人は送信した事実を否認できないという特徴を持っている

- 処理が遅い

  公開鍵暗号方式は秘密鍵暗号方式に比べて速度がきわめて遅くなるので、大量のデータを送ることには向いていない。公開鍵暗号の有力な利用法は秘密鍵暗号の鍵を送信することである。

- 相手認証は不可能

  秘密鍵暗号方式では、当事者しか知らない秘密鍵を用いて暗号通信するため、暗号文が正しく複合できるかどうかによって通信相手を認証できる。しかし公開鍵暗号方式では公開鍵を用いて誰でも暗号通信できるため、受信者は誰から暗号文が送られてきたかがわからない。（電子署名の機能を使えば、相手認証が可能である。）

第4章　実験

ReliableSocket を組み込んだ JavaRMI を利用して音楽再生アプリケーションを作成した。このアプリケーションはサーバがネットワーク経由で音楽データを配信し、クライアントが受信した音楽データを再生するというものである。

サーバが音楽データを配信している最中にクライアント側でネットワークを切断し、しばらく時間をおいてネットワークに再接続したら続きから音楽が再生できることを確かめた。また、何度切断しても正しく動作することも確かめた。このアプリケーションプログラムは付録に載せる。

次に、ReliableSocket のスループットと遅延時間、RMI で ReliableSocket を使用するときのリモートメソッド呼び出しにかかる時間を測定する実験を行った。

実験に用いた計算機は、Sun Fire V480(UltraSPARC-III Cu 900MHz x 4, 16GB, Solaris 9 4/03)、(Pentium 4 1.92GHz, 384MB, Windows XP Professional Version 2002) である。JVM は Sun JDK 1.4.0 を用いた。ネットワーク速度は 100Mbps である。

4.1　Throughput

8M のデータを 8bytes から 32KB までの異なるブロックサイズで送信してスループットを計算した。ブロックサイズとは送信するデータのサイズのことである。

データを送信する直前からデータが全て送信して相手に届いたことが確認できるまでの時間を計ることでスループットを計算した。

結果は以下の表のようになった。

この結果から、Socket,ReliableSocket ともにブロックサイズが大きいほどスループットが大きくなることがわかる。これは、ブロックサイズが大きくなるほど、ネットワークを使用する回数が減るためである。また、ブロックサイズが大きくなるほど Socket と ReliableSocket の差が少なくなる。この理由はブロックサイズが小さいときは頻繁にネットワークを使用するために ReliableSocket のオーバーヘッドが目立ってしまうからである。

表 4.1: Socket と ReliableSocket のスループット

BlockSize	Socket(Mb/s)	ReliableSocket(Mb/s)
8 bytes	6.15	0.504
16 bytes	8.95	0.94
32 bytes	9.60	1.91
64 bytes	9.13	3.51
128bytes	10.29	5.25
256bytes	10.16	7.11
512bytes	10.49	7.98
1KB	13.0	9.44
2KB	13.23	9.67
4KB	16.9	9.33
8KB	16.1	9.55
16KB	16.34	9.02
32KB	17.2	8.91

4.2　遅延時間

8M のデータを 8bytes から 1KB までの異なるブロックサイズで送受信を行った。送受信を行う直前から送受信が終わるまでの時間を計ることで遅延時間を計算した。

結果は以下の表のようになった。

表 4.2: Socket と ReliableSocket の遅延時間

BlockSize	Socket(msec)	ReliableSocket(msec)
8 bytes	0.87	1.13
16bytes	0.86	1.14
32 bytes	0.87	1.13
64 bytes	0.93	0.26
128bytes	1.54	1.94
256bytes	1.72	2.04
512bytes	1.96	2.36
1KB	2.52	3.17

この結果から Socket と ReliableSocket では遅延時間に 0.5msec 前後の差があることがわかる。ReliableSocket のオーバーヘッドは送信するデータを保存する処理とセレクタの処理にある。

4.3　JavaRMI でのリモートメソッド呼び出しにかかる時間

JavaRMI で Socket を使った場合と、ReliableSocket を使った場合のリモートメソッド呼び出しにかかる時間を引数のサイズを変えて実験した。

実験結果は以下の表のようになった。

表 4.3: 1 回あたりのリモートメソッド呼び出しにかかる時間

引数	Socket(msec)	ReliableSocket(msec)
8bytes	1.43	2.79
16bytes	2.09	2.86
32bytes	2.83	3.325
64bytes	2.94	3.435
128bytes	2.87	3.415
256bytes	2.84	3.525
466bytes	2.93	3.7
467bytes	3.38	198.89
1KB	2.64	199.89
8KB	4.91	200
1MB	281	339
8MB	1962	2323

この結果から、ReliableSocket を使用すると、467bytes のときに急激に時間がかかるようになっている。ReliableSocket の実装にバグがあるのかもしれないが、どこに原因があるのか確かめることができなかった。ReliableSocket は内部で SocketChannel を使用しているので RMI との相性があまり良くないということも考えられる。

第5章　まとめと今後の課題

本稿では Java 用分散ミドルウエアを高信頼化するためにソケットを高信頼化（ReliableSocket）することを提案した。ReliableSocket を利用することで、さまざまな分散ミドルウエアを容易に高信頼化することが可能である。本研究では実際に JavaRMI に ReliableSocket を組み込んで高信頼化できていることを確かめた。JavaRMI は現在広く使われている分散ミドルウエアであり、カスタム RMI ソケットファクトリの作成によりソケットを差し替えることができる。

モバイルコンピューティングの環境では場所を移動するたびにネットワークの物理的な切断や IP アドレスの変更といったことが行われる。ネットワークの物理的な切断や IP アドレスの変更が行われると TCP コネクションは失われてしまう。そのため分散アプリケーションを利用してリモートホストと通信をしている場合は移動するたびにアプリケーションが中断してしまうことになる。

ReliableSocket は通常の Java のソケットに失われたコネクションの自動的な回復機能と失われたデータの再送機能を加えたものである。ReliableSocket を組み込んだ分散ミドルウエアを利用してアプリケーションを作成すればユーザは場所を移動するときにいったんネットワークを切断しても移動先でふたたびネットワークに再接続すればアプリケーションを続けて使用することが可能となる。また無線 LAN で通信をしていて, 場所の移動によってクライアント側の IP アドレスが変更される場合でも、サーバが移動しない限りはコネクションの回復を行うことができる。

今後の課題は再接続時に認証を行うことである。認証が実装できていないために再接続してきた相手が今まで通信していた相手であることを確かめることができない。また、現在の実装では場所を移動するのはクライアント側だけを想定している。そのためサーバが移動して IP アドレスが変更してしまう場合はコネクションの回復を行うことができない。サーバが移動した場合でもコネクションの回復を行えるような実装のほうが望ましい。

付 録A　デモで使用したアプリケーションプログラム

リモートインターフェース

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import javax.sound.sampled.AudioFormat;
public interface Music extends Remote {
    public AudioFormatRec getAudioFormat() throws RemoteException;
    public byte[] getNextData() throws RemoteException;
}
```

音楽を配信するサーバプログラム

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.io.File;
import java.io.IOException;
import java.io.Serializable;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import javax.sound.sampled.AudioFormat;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.UnsupportedAudioFileException;
import javax.sound.sampled.AudioFormat.Encoding;
public class MusicServer implements Music{
    private final int BUFFER_SIZE = 128000;
    private File soundFile;
    private AudioFormat audioFormat;
    private AudioInputStream audioInputStream;
```

```
    public static void main(String args[]) throws Exception{
        System.setSecurityManager(new SecurityManager());
        System.out.println("start server");
        try {
            MusicServer obj = new MusicServer("sample.wav");
            /*
             *ReliableSocket を使うように設定
             */
            RMIClientSocketFactory csf = new ReliableClientSocketFactory();
            RMIServerSocketFactory ssf = new ReliableServerSocketFactory();
            Music stub =
                (Music) UnicastRemoteObject.exportObject(obj, 8000, csf, ssf);
            LocateRegistry.createRegistry(2002);
            Registry registry = LocateRegistry.getRegistry(2002);
            registry.rebind("Music", stub);
            System.out.println("MusicServer bound in registry");
       } catch (Exception e) {
            System.out.println("MusicServer exception: " + e.getMessage());
            e.printStackTrace();
       }
    }

    public MusicServer(String file) throws UnsupportedAudioFileException,
IOException{
        // File クラスのインスタンスを生成
        soundFile = new File(file);
        // オーディオ入力ストリームを取得
        audioInputStream = AudioSystem.getAudioInputStream(soundFile);
        // オーディオ形式を取得
        audioFormat = audioInputStream.getFormat();
       }

    public AudioFormatRec getAudioFormat() throws RemoteException {
       return new AudioFormatRec(audioFormat);
    }

    public byte[] getNextData() throws RemoteException {
        byte[] buff = new byte[BUFFER_SIZE];
```

```
        try {
            int n = audioInputStream.read(buff);
            if(n < 0){
                audioInputStream = AudioSystem.getAudioInputStream(soundFile);
                n = audioInputStream.read(buff);
            }
            byte[] tmp = new byte[n];
            System.arraycopy(buff, 0, tmp, 0, n);
            System.out.println("transport wav data :"+n+" byte  time:"+
System.currentTimeMillis());
            return tmp;

        } catch (Exception e) {
            return new byte[0];
        }
    }
}
```

音楽を再生するクライアントプログラム

```
import java.rmi.*;
import java.rmi.registry.*;
import javax.sound.sampled.AudioFormat;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.DataLine;
import javax.sound.sampled.SourceDataLine;
public class MusicClient {
    public static void main(String args[]) throws Exception{
        // セキュリティマネージャーを設定します
        System.setSecurityManager(new SecurityManager());
        try {
            Registry registry = LocateRegistry.getRegistry(2002);
            Music obj = (Music) registry.lookup("Music");
            AudioFormat audioFormat = obj.getAudioFormat().getAudioFormat();
            // データラインの情報オブジェクトを生成
            DataLine.Info info = new DataLine.Info(SourceDataLine.class,
```

```
audioFomat);
            // 指定されたデータライン情報に一致するラインを取得
             SourceDataLine line = (SourceDataLine) AudioSystem.getLine(info);
            // 指定されたオーディオ形式でラインを開く
            line.open(audioFormat);
            // ラインでのデータ入出力を可能にする
            line.start();

            int n=0;
            byte[] data = obj.getNextData();
            while(data.length > 0){
                line.write(data, 0, data.length);
                data = obj.getNextData();
                System.out.println("get wav data "+data.length+" byte  time:"+
System.currentTimeMillis());
            }
            line.drain();
            line.close();
            System.exit(0);
        }
        catch(Exception e){
        }
    }
}
```

AudioFormatRec クラス

```
import java.io.Serializable;
import javax.sound.sampled.AudioFormat;
public class AudioFormatRec implements Serializable {
    float sampleRate;
    int sampleSizeInBits;
    int channels;
    boolean signed = true;
    boolean bigEndian;
    public AudioFormatRec(AudioFormat audioFormat) {
        sampleRate = audioFormat.getSampleRate();
        sampleSizeInBits = audioFormat.getSampleSizeInBits();
```

```
        channels = audioFormat.getChannels();
        AudioFormat.Encoding enc = audioFormat.getEncoding();
        System.err.println(enc.toString());
        if (enc.toString().equals("PCM_UNSIGNED")) { signed = false; }
        bigEndian = audioFormat.isBigEndian();
    }

    public AudioFormat getAudioFormat(){
        return new AudioFormat(sampleRate, sampleSizeInBits, channels, signed,
bigEndian);
    }
}
```