第 13 部

自動翻訳

第 1 章

序論

1.1 背景

コンピュータネットワークというものは、この度の PACCOM の開通でもわかるように、現在、日本の中だけで行われているものではなく、日本とアメリカというように国と国の間で通信を確立するために使用されているものである。国家間での通信を行うということは、当然、異種の言語どうしでのコミュニケーションが必要になってくる。そこで考えられるのは、ネットワーク上において機械翻訳システムを使えるようにし、異種の言語間でのコンピュータコミュニケーションを容易に行えるようにすることである。

ネットワークで機械翻訳システムを使えるようにするためには、まずそのシステムを複数のユーザ（ホスト）が共有できるようなものにしなければならない。つまり、自動翻訳（機械翻訳）サーバ：Automatic Translation Server（ATS）とそれを使用するためのクライアント：Automatic Translation Client（ATC）を設計し、ネットワークにおける自動翻訳機構を構築しなければならない。これが実装されると、例えば日本語の文章を英語に翻訳して日本からアメリカに mail を送ったりして通信を行うことができる。また、機械翻訳システムを talk や phone に組み込んで、直接日本とアメリカで通信を行うことも可能である。本論文では、自動翻訳サーバ（ATS）とそれを使用するためのクライアント（ATC）を設計し、これらを実装し、ネットワーク上で効率的に機械翻訳を行えるようにすることを目標とした。

1.2 本研究の目的

現在の機械翻訳システムは、単なるアプリケーションプログラムで、データの入出力から翻訳までを単一のホストで行うようになっている。よってこのままでは、それぞれのホストに機械翻訳システムを稼働させて、し

かも単一のホスト内で機械翻訳を行わなければならない。ネットワークを利用できる環境で全てのホストにそれぞれ ATS を稼働させるのは、資源の無駄使いであり、ネットワークの有効利用をしていないことになる。そこで、この機械翻訳システムを分散環境で利用できるサーバ型に改良して、複数のユーザ（ホスト）が、あるいくつかのホストで稼働するサーバ型の機械翻訳システムを共有できるようにする必要がある。これによって、各ホストのコンピュータ資源の消費を軽減し、広域分散環境においても機械翻訳がより効率的に行えるようにすることができる。

自動翻訳サーバ（ATS）とそれを使用するためのクライアント（ATC）を設計する際に一番重要なことは、ホストとホストの間でのデータのやりとり、即ちクライアントとサーバ間のプロトコルをどのようにするかを決めることである。

1.3 本論文の構成

本論文の構成は、第１章で本研究の背景と目的について述べる。第２章で機械翻訳システムについて述べる。第３章では、プロセス間通信（IPC）について考え、サンプルプログラムをもとに、インターネットでの通信を中心に、クライアント／サーバモデルについて、バーチャルサーキット型とトランザクション型の特徴と違いや問題点を述べる。第４章では、広域分散型機械翻訳システムを構築するにあたって、自動翻訳のためのクライアント（ATC）と自動翻訳サーバ（ATS）と ATC を ATS に割り当てるためのスケジューリングサーバ（SS）をもとに考えられる６つのシステムモデルについて考察する。第５章でこの翻訳システムを実際にネットワーク上に構築した際に、どのような通信の流れをなすか、またどのような特徴と問題点があるかを考える。第６章でこの翻訳システムの評価を行い、その結論と今後の課題を述べる。最後に謝辞及び参考文献をまとめる。

第 2 章

機械翻訳システム

ここでは、機械翻訳とその応用について述べる。

2.1 機械翻訳とは

コンピュータが世の中に登場して以来、科学技術計算を中心にいろいろな面でそれが役に立っている。しかし、コンピュータに計算能力だけではなく人間のように思考する知的な能力を持たせることができないかということは、昔から現在に至るまで多くの研究者が考えてきた。その中の１つとして上げられるのが、機械翻訳である。

現在、機械翻訳システムはいくつかの企業から商品として提供され、実用化されている。しかし、人間が行っても困難な翻訳をコンピュータにやらせようとするのだから、当然解決しなければならない課題も少なくない。これを克服するために、今もなお多くの研究者が機械翻訳に取り組んでいる。

2.1.1 機械翻訳の仕組み

機械翻訳とは、入力文（原文）を解析し、コンピュータで処理できる内部表現（中間表現）を得たあと、その内部表現から翻訳対象言語の文（訳文）を生成する処理である。この処理を翻訳処理という。翻訳処理では、言語に関する知識として辞書と文法を使う。

原文が書かれている言語を原言語と呼び、訳文を書くための言語を目標言語と呼ぶ。翻訳処理では、原文の解析は原言語について行い、訳文の生成は目標言語について行う。

原文の解析は、翻訳しようとしている文章の意味内容を辞書と文法を使って分析しその結果を計算機内部に表現する。このように文を解析して得られた文の構造を計算機内部で表現したものを中間表現と呼ぶ。この中間表現の内容は原言語の性質を反映している。従って、この中間表現から訳文を生成するためには、これを目標言語の性質を反映する内容に直さな

ければならない。そのための処理が中間表現の変換であり、原言語と目標言語を結び付ける辞書と文法を使って行う。

機械翻訳の処理には、大きく分けて２つの方式がある。１つは上記のように中間表現の変換を必要とするもので、これを変換方式またはトランスファ方式と呼ぶ（Figure2.1参照）。中間表現を言語に依存しない内容で表しておくと、中間表現の変換は必要なくなる。このように、中間表現の変換を行わないものを、中立言語方式またはピボット方式と呼ぶ（Figure2.2参照）。一般に、中間表現が構文的な内容を表すときはトランスファ方式となり、意味的な内容を表すときはピボット方式となる。これらの処理形態を簡単に示すと、次のようになる。目標言語に関する辞書と文法を使って、この変換された中間表現から訳文を生成する。

1. 構文トランスファ方式（構文的な構造を中心に解析）
   (a) 入力文の解析
   (b) 入力文を記述している言語の性質を反映する中間表現の作成
   (c) 訳出文を記述している言語の性質を反映する中間表現への変換
   (d) 訳出文の生成
2. 意味ピボット方式（構文的な構造と文の意味内容をもとに解析）
   (a) 入力文の解析
   (b) 入力文を記述している言語の性質を反映する中間表現の作成
   (c) 入力文を記述している言語の性質を反映する中間表現の解析
   (d) 言語に中立な内容を示す中間表現の作成
   (e) 訳出文を記述している言語の性質を反映する中間表現の生成
   (f) 訳出文の生成

2.1.2 これからの機械翻訳システム

１９７０年代に構文トランスファ方式のシステムが出現するようになった。構文解析の方法は、大きく分けて句構造文法と依存文法がある。句構造文法は、名詞句や動詞句などの句単位に文の構造をまとめる文法である。また依存文法は、語と語の依存関係によって文の構文構造を与える。両方とも表現能力は同じであることが証明されている。しかし、システムの検証が進むにしたがって、構文の理解だけでは言語が持っているあいまいさを解決するには十分でないということが分かってきた。そこで、１９８０

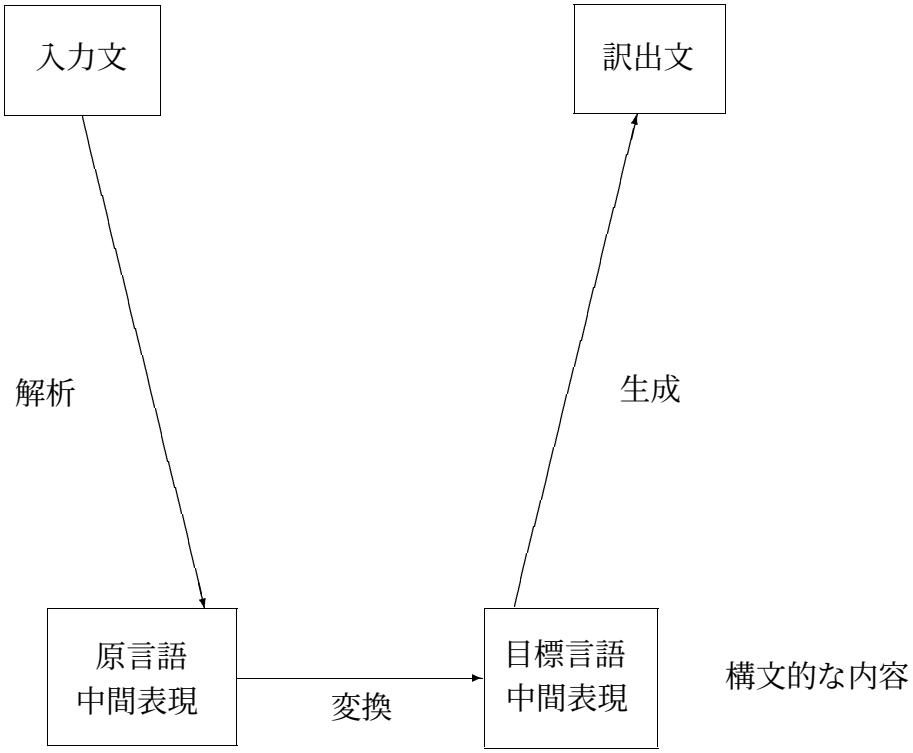

図 2.1: 構文トランスファ方式

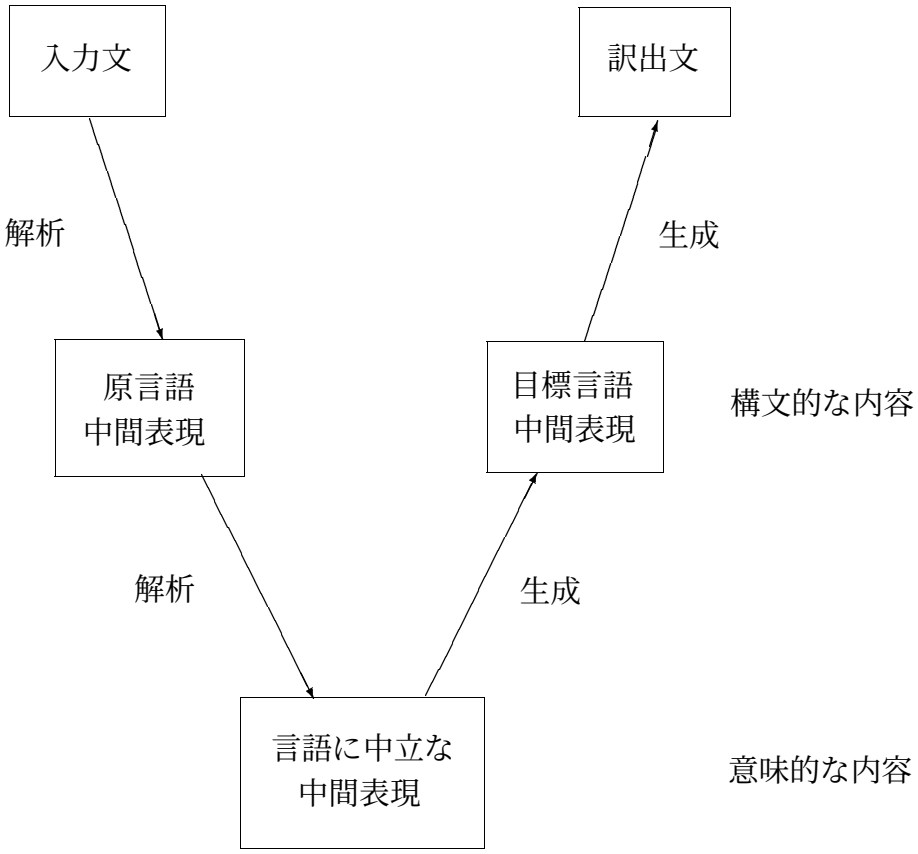

図 2.2: 意味ピボット方式

年代に意味理解を重視したピボット方式のシステムが出現するようになった。これは、日本の企業が中心となっている。なぜならば、英語や仏語よりも日本語の方が意味理解をより重要としているからである。意味理解を行うためには、人が言葉を理解するために用いているような知識や常識を必要とする。つまり、意味理解のための知識ベースを確立しなければならない。ところが、この意味に関する情報を収集するには莫大な労力と時間が必要なので、今もまだ発展している段階である。

機械翻訳システムの技術は、かなり進歩してきているが、まだ解決されるべき課題も多い。完璧な翻訳システムを目指すことは大変難しい。しかし、何百万もの専門用語を記憶しておくなど、人間よりもコンピュータの方が適しているようなこともいくつかある。従って機械翻訳は、まず第一に計算機でなければできないような処理を行うようにし、翻訳者を支援することが目的である。この考え方は、これから先も変わることはないだろう。

2.2　ネットワークにおける機械翻訳

すでに述べたように、ネットワーク上で機械翻訳システムを効率よく利用するためには、このシステムをサーバ型に改良しなければならない（Figure2.3参照）。通信という観点からネットワークを眺めてみれば、この自動翻訳サーバ（ATS）の設計及びこれによる翻訳機構というものは、いろいろな意味でこれからのネットワークを支える基盤となるものであることは言うまでもない。

2.2.1　多言語間の機械翻訳

世界中をコンピュータネットワークで張り巡らしている現在においては、それを使って国際的にコミュニケーションを行おうとすることは自然なことである。それにはまず、できる限り多くの言語の機械翻訳システムがあることが望ましい。しかし、そのシステムを開発するとなると、多大な労力を必要とする。また、翻訳者が少ない言語のシステムを開発するのも大変である。しかし、言語間で共有できる中間表現や知識を少しずつ収集していけば、より多くの言語の機械翻訳システムを作り上げることも可能になるだろう。

2.2.2　talk や phone への機械翻訳システムの応用

我々はコンピュータネットワークにおいて、端末上で会話を行うときに、talk や phone を使う。talk は二人のユーザが会話をするときに使用し、

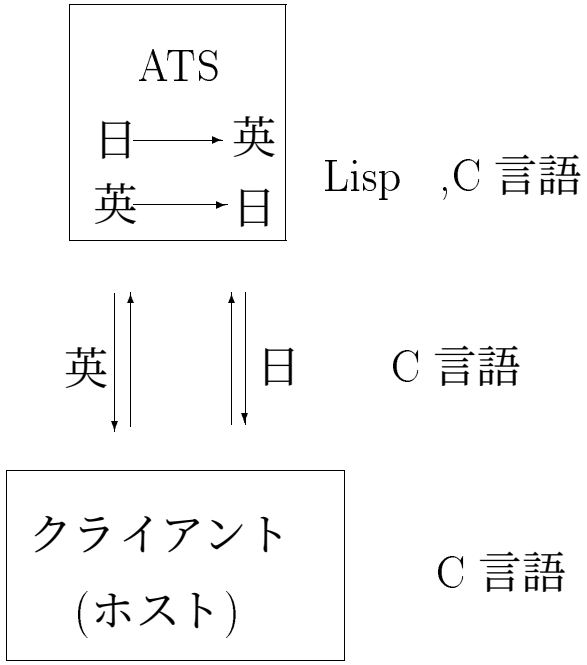

図 2.3: 自動翻訳機構

phone は二人以上のユーザが会話をするときに使用する。通常の phone はローマ字かひらがなでメッセージの交換を行うが、これに機械翻訳を組み込んで日本語のメッセージを英語に変換して送ったり、英語のメッセージを日本語に変換したものを受け取ったりして会話を行うことを考える（Figure2.4 参照）。これが実現されれば、国家間でのコンピュータコミュニケーションもやりやすくなる。

しかし、実際にはかなり多くの問題が考えられる。最大の問題は、通信時間のことである。本研究で使用する機械翻訳システムは、株式会社 CSK で開発された SUN workstation 上で動作するもので、Lisp によって書かれている。終止符（。）で終わるまでの一文を入力して結果を得るまでにやや時間がかかる。通信時間が限りなく 0 に近づくことを望まれる talk や phone では、これは深刻な問題である。解決法としては、まず翻訳システムそのものを改良することが考えられる。ところが、これは長年の技術と経験があってこそできることなので、ここでは別な方法を利用することに着目する。ソフトウェアからのアプローチではなく、ハードウェアからのアプローチを考える。つまり、高速の CPU を持つマシンに翻訳システムを乗せることによってこの遅さをカバーしようとするのである。自動翻訳サーバ（ATS）を高速の CPU の上で実装することによってどれだけ遅さがカバーされるかについては、実際に試してみないとわからないが、金銭の問題をあまり考えなければ、これが翻訳処理時間の短縮につながることは言うまでもない。

他の問題点としては、日常会話を十分に扱えるほど翻訳機構が整備されていないことや、翻訳できない部分を含むような文章をどのように扱ったらいいかとか、一文全体を翻訳し終えるまで相手に送るのを待たなければならないのかということが挙げられる。科学技術に関する文章を翻訳するのは割と容易であるが、さまざまな言い回しがある日常会話を翻訳することは容易ではない。これは talk や phone に限らず機械翻訳全体における課題である。まず、辞書の整備から行わなければならないだろう。また、翻訳できない部分が存在する文章は、ユーザの手直しを必要とする。つまり、一度サーバから送信者にその文章が返されて、完全に翻訳しなおしたら、受信者に再転送される。翻訳は一文全体が揃わないと行えないので、文単位で相手に転送することになる。

これらの問題点を考えると、Translation Phone の実現はかなり難しいが、これは将来のネットワークにおいて絶対に必要なものであると思う。

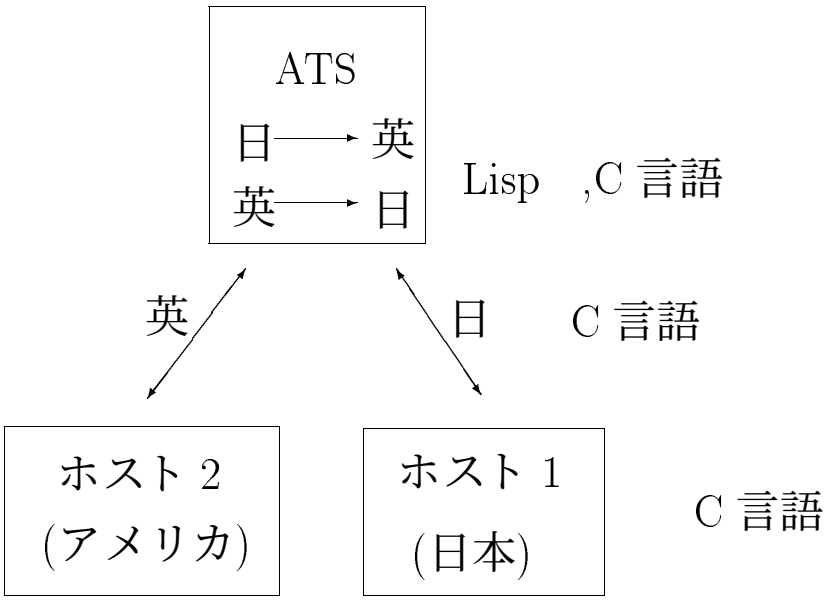

図 2.4: Translation Phone

第 3 章

プロセス間通信（IPC）

コンピュータネットワークのもとでは、あるホストで動作しているいくつかのサービスを提供するプロセス（プログラム）に、別のホストからデータを送信してその結果を受信するようなことがある。このデータを送受信するものも１つのプロセスである。このように、あるプロセスと別のプロセスの間で行われるようなデータの送受信をプロセス間通信（interprocess communication）：IPC [63] と呼ぶ。UNIX ４．３ BSD は IPC のためのいくつかの方法を提供している。ここでは、インターネットドメインのストリーム転送とデータグラム転送を中心に述べる。

3.1 ドメインとプロトコル

プロセスとプロセスの間でコミュニケーションを行おうとするとき、それらが独立に生成されたものであるならば、どのようにすればよいのだろうか。ここで考えられるのが、２つのプロセスがそれぞれにソケットを生成し、それらの間でデータメッセージを送受信することである。UNIX ４．３ BSD では、個々にソケットを生成し、それらに名前を与えて、それらの間で通信を行うことをサポートしている。

各々のプログラムで生成されたソケットは、お互いが問い合わせをするための名前を使用する。その名前は、使用されるときにはネットワークアドレスに変換される。このようなアドレスが記述されている空間をドメイン（domain）と呼ぶ。ソケットのためのドメインはいくつかあるが、その中でよく知られているものが次の２つである。

- UNIX ドメイン
- インターネットドメイン

UNIX ドメインでは、ソケットは、ファイルシステムの名前空間内でパスネームを与えられる。ファイルシステムノードがソケットのために生成

されて、他のプロセスは、そのパスネームを与えることによってソケットに問い合わせをする。よって、UNIX ドメインは、同一ホスト内で動作するプロセス間での通信を提供する。インターネットドメインは、DARPA インターネットの標準プロトコルである IP ／ TCP ／ UDP の UNIX 上での実装である。インターネットドメインでのアドレスは、ネットワークアドレスとポートと呼ばれる識別番号から成る。よって、インターネットドメインは、ホストとホストの間での通信を提供する。つまり、異なるホストで動作するプロセス間での通信を提供する。

通信の形態には次の 2 つの種類がある。

- ストリーム型
- データグラム型

ストリーム型の通信は、2 つのソケットの間でコネクションをつないで行う。これは、信頼性があり、送信するデータに誤りがあるかどうかを気にする必要がなく、メッセージデータに境界がない。送られてくるデータが大きすぎて全部を受信できるほどのメモリーが空いていないときは、データの一部だけを読むこともできる。ストリーム型の通信を実装するプロトコルは、エラーとともに受信したメッセージを再転送する。また、コネクションが切れたあとでメッセージを送ろうとすると、エラーメッセージが返ってくる。データグラム型の通信は、コネクションを使用しない。各々のメッセージは、直接送り先のアドレスを指定して送信される。そのアドレスが正しければ、一般には相手に受信されるが、その保証はされていない。従って信頼性がないのである。データグラム型の通信は、受信者からの応答を求めてくるような要求を出すために使われることがよくある。適正な時間内にその応答が到着しなければ、その要求は繰り返される。また、データグラム型のメッセージには境界が存在し、読まれるときは分割されている。

ストリーム型の通信において、コネクションを確立するためのコンピュータ資源の消費は、コネクションを頻繁に使用してはじめて正当化される。また、データグラム型の通信において、失われたメッセージをただ無視していると、トラフィックの量は重大な問題になる。プロトコルの動作は、ネットワークの状態が変わると、それにつれて変化するから、プロトコルの動作が大きく影響を及ぼすようなプログラムでは、最近のネットワークの情報を得て、それによってプロトコルを選択することが望まれる。

プロトコル（protocol）とは、データフォーマットや通信関係者の間でのデータ転送に関する決まりというような一連の規則である。一般には、各々のドメイン内で、各々のソケットの型に対して 1 つのプロトコルがあ

る。プロトコルを実装しているコードは、ソケットにバインドされる名前のトラックを保持している。その名前によって、コネクションを確立し、ソケットの間でデータ転送を行う。いくつかのプロトコルが、ある特定のドメイン内で同じ通信の型を実装することは可能である。また、使用するプロトコルを選択することは可能であるけれども、ほとんどすべての使用において、デフォルトのプロトコルを要求すれば十分である。インターネットドメインでは、ストリーム型の通信においては TCP ／ IP を使用し、データグラム型の通信においては UDP ／ IP を使用する。

3.2　クライアント／サーバモデル

現在の機械翻訳システムは、それが動作しているホストからデータを入力して翻訳された結果をそのホストに出力する。ネットワークが確立されている以上は、この翻訳システムをあるホストで動作させておいて、別のホストからデータを入力して翻訳された結果をこのホストに出力されるようにしたいものである。つまり、翻訳というサービスを提供するプロセスとデータの入出力（送受信）を行うプロセスとの間の通信を確立させるわけである。このように、いくつかのサービスを提供し、ネットワークを通して到着した要求を受け入れて、そのサービスを実行し、要求を出したものに結果を返すようなプログラムをサーバ（server）と言い、このサーバに要求を送り、応答を待つことを行うプログラムをクライアント（client）と言う。クライアント／サーバモデルの動作原理は、次のようになる。

1. サーバを起動する
2. サーバはクライアントからの要求を待つ
3. クライアントからサーバへ要求を送信する
4. サーバはクライアントから要求を受信する
5. サーバは子プロセスを生成する
6. サーバの子プロセスはその要求を処理し、プロセスを終了する
7. クライアントはサーバの子プロセスからの応答を受信する
8. サーバの親プロセスは再びクライアントからの要求を待つ

クライアント／サーバモデルのサンプルプログラムとして、英小文字から英大文字への変換と英大文字から英小文字への変換とローマ字からひら

がなへの変換の３つのサービスを提供するサーバプログラムと、このサーバにデータを送信して変換された結果を受信して出力するクライアントプログラムを作成した（付録１参照)。バーチャルサーキット型とトランザクション型の２種類があるが、これについての詳しいことは３．３で述べることにする。

通常サーバは、提供するサービス用に予約されているポート番号を使用する。ポート番号は、ソケットの識別子のために使われる。しかし、この作成したサーバは予約されているポート番号を持つサービスを提供しないので、予約済みでないポート番号を使わなければならない。まず始めは０のポート番号を指定しておく。それからソケットにバインドすると、OSが適当なポート番号を割り当ててくれる。これによってサーバは動作を開始し、クライアントの要求を待つ。

クライアントは、このサーバが動作しているホスト名と指定されたポート番号を使ってサーバに要求を出し、応答を待つ。サーバはクライアントからの要求を受信したら、ストリーム転送のサーバにおいては、フォークして子プロセスを作り、これに要求の処理をさせて、親プロセスは別のクライアントからの要求を待つ。データグラム転送のサーバにおいては、シーケンシャル処理なので、順番に要求を受信し処理していく。クライアントは結果を受信したら、また要求を出すか動作を終了する。すべてのクライアントが動作を終了しても、サーバは動作を終了することなく永久にクライアントからの要求を待つ。サーバは、強制的に止められる以外は動作を終了することはない。

3.3 バーチャルサーキット型とトランザクション型

インターネット標準のトランスポートレベルプロトコルには、２つのプロトコルが存在する。１つは、信頼性のあるストリーム転送を提供するもので、TCP（Transmission Control Protocol）[14] と呼ばれる。もう１つは、信頼性のないデータグラム転送を提供するもので、UDP（User Datagram Protocol）[14] と呼ばれる。バーチャルサーキット型は、TCP によるストリーム転送を使用し、トランザクション型は、UDP によるデータグラム転送を使用する。(Figure3.1、Figure3.2参照）これらの動作原理を簡単に示すと、次のようになる。

1. バーチャルサーキット型

   (a) クライアントからサーバへ使用したいサービス番号を送信する

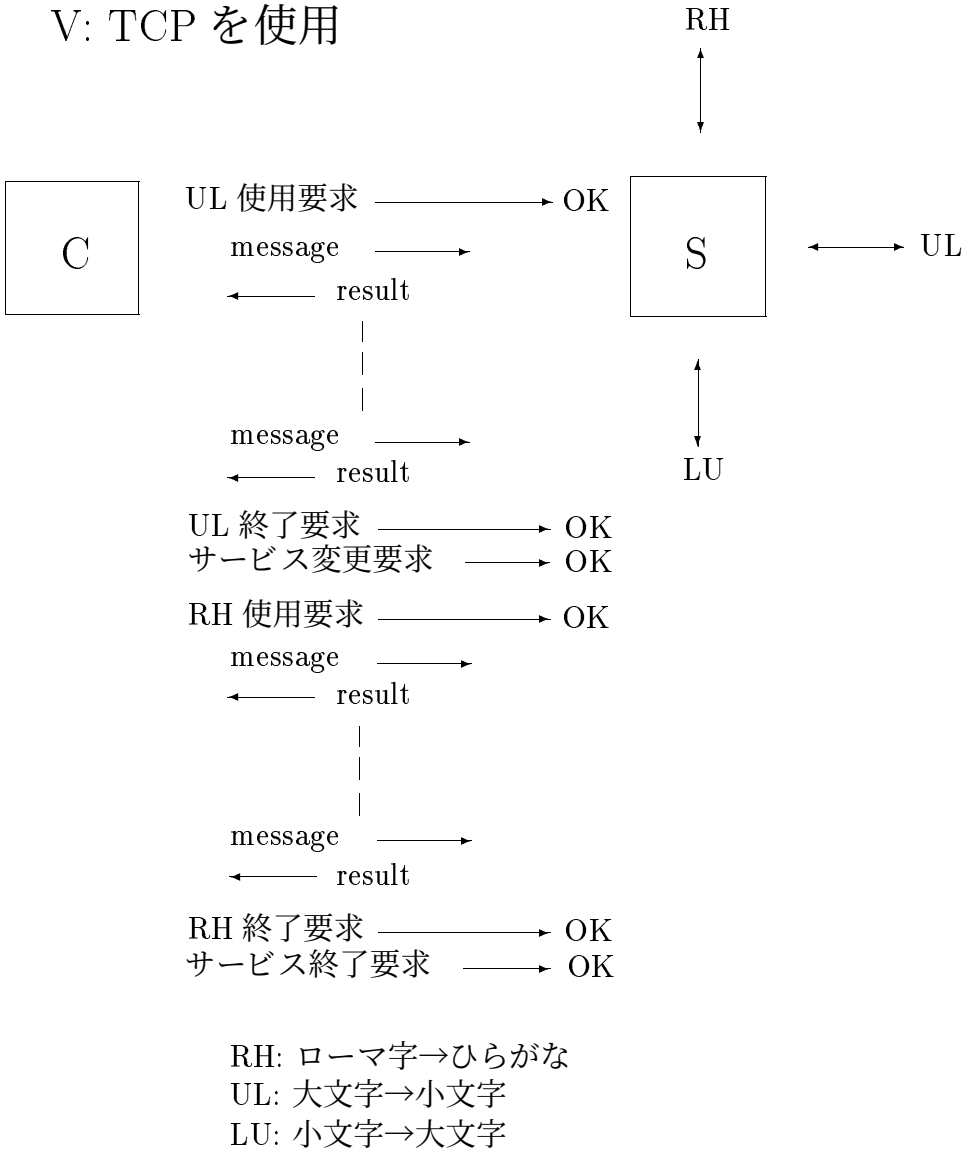

図 3.1: バーチャルサーキット型

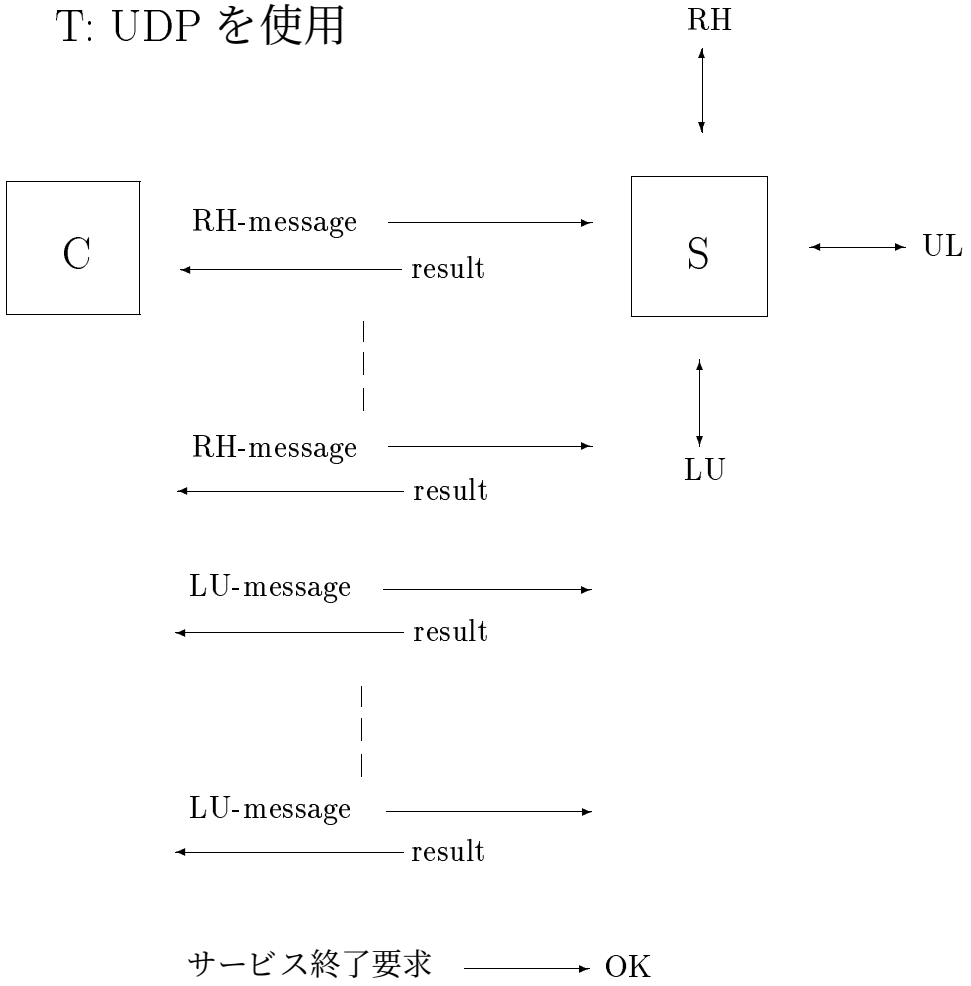

図 3.2: トランザクション型

(b) サーバはそれを受信したら、クライアントからのデータメッセージを待つ

(c) クライアントはサーバにデータメッセージを送信する

(d) サーバはそれを受信したら、処理して、クライアントに送り返す

(e) あとは、このデータメッセージの送受信だけを繰り返す

2. トランザクション型

(a) クライアントからサーバへ使用したいサービス番号とデータメッセージをいっしょに送信する

(b) サーバはそれを受信したら、サービス番号に従って処理して、クライアントに送り返す

(c) あとは、このサービス番号＋データメッセージの送受信を繰り返す

ここでは、このクライアント／サーバモデルの２つの形式について詳しく述べることにする。なおこれから先においては、バーチャルサーキット型のことを V 、トランザクション型のことを T と書くことにする。

3.3.1　インターネットド メインでの通信

インターネットド メインのアドレス構造体は、＜netinet/in.h＞ファイルに定義されている。ソケットのアドレスはホストのアドレス（３２ビット）とそこでのポート番号を明確にする。ポート番号は特定のプロトコルを実装しているシステムルーチンによって管理されている。ホスト間でメッセージデータが送られるとき、それはまずデスティネーションホストのプロトコルルーチンに送られて、そのプロトコルルーチンがメッセージを送るべきソケットを決定するために、いっしょに送られてくるデスティネーションアドレスを解析する。いくつかの異なるプロトコルが同じホストで稼働しているが、それらが同じポート番号を使用することは許されている。そこで、ソケットのアドレスは、ホストアドレスとポート番号とプロトコルタイプの３つから構成される。ソケット間で通信を行うとき、相手を明確にするために、プロトコルタイプ、ローカルホストアドレス、ローカルポート、リモートホストアドレス、そしてリモートポートを使用する。データグラムソケットを使用する場合、相手を明確にするものは、send 命令が作用している間だけ必要になる。

ソケットが生成されるとき、それのためのプロトコルが選択される。クライアントから見て、サーバのソケットのためのローカルホストアドレス

は、明確なホストのネットワークアドレスが必要である。サーバの動いているホストのアドレスを得るために、サンプルプログラムのサーバでは、gethostname() を呼んでホスト名を得て、それから gethostbyname() を呼んでホストのアドレスを得ている。これを用いて、ソケット用の構造体のアドレスをセットする。また、サーバ自身は、必ずしも自分の動いているホストのアドレスを知る必要はないので、ソケット用の構造体のアドレスをワイルドカードである INADDR_ANY（０）にセットすることもできる。サンプルプログラムのサーバを起動させるときに、オプション a をつけるとこの場合になる。アドレスが不確定のところからデータを受け取ることは可能であるが、アドレスが不確定のところにデータを送ることはできない。従って、クライアントの方は、まずコマンド行の引数からデスティネーションホストの名前を得て、メッセージを送るためのネットワークアドレスを決定するために、サーバと同様に gethostbyname() を呼んでホストのアドレスを調べる。そして、ホストのネットワークアドレスを含む構造体が返されて、そのアドレスがメッセージのデスティネーションを明確にするための構造体へコピーされる。

ポート番号（port number）とは、郵便箱のような数字であって、プロトコルはそこにメッセージを置く。公示されているサービスを提供するあるデーモンは、あらかじめ予約されたポート番号を持っている（Table3.1、Table3.2参照）。これらは、１から１０２３までに予約されている。この２つの表を見てもわかるように、前で述べたような、TCP と UDP の２つのプロトコルが同じポート番号を使用していることが示されている。これより大きな数字が、一般のユーザが利用できる数である。サーバだけが特定の数を要求する必要がある。アドレスがソケットにバインドされるとき、システムが未使用のポート番号を割り当てる。ソケット用の構造体に０のポート番号を入れておいてバインドするときか、バインドされていないソケットにコネクトしたり send したりするときにこれが起こる。ポート番号は自動的にユーザに知らされるわけではないから、ポートを０にして bind() を呼び出したあと、getsockname() を呼び出して、実際に割り当てられたポート番号を調べなければならない。getsockname() によって返される値はネットワーク表現の数なので、これをホスト表現の数に変換しなければならない。サンプルプログラムでは、ntohs() というルーチンを呼んでこれを行っている。

ストリーム型のソケットの間でデータ交換を行うときは、そのソケットの間にコネクションをつながなければならない。クライアントの方は、コネクションを確立するために、ストリーム型のソケットを生成したら、コネクトしたいソケットのアドレスを明確にして、connect() を呼び出してい

Decimal	Keyword	Description
0		Reserved
1-4		Unassigned
5	RJE	Remote Job Entry
7	ECHO	Echo
9	DISCARD	Discard
11	USERS	Active Users
13	DAYTIME	Daytime
15	NETSTAT	Who is up or NETSTAT
17	QUOTE	Quote of the Day
19	CHARGEN	Character Generator
20	FTPDATA	File Transfer Protocol(data)
21	FTP	File Transfer Protocol
23	TELNET	Terminal connection
25	SMTP	Simple Mail Transport Protocol
37	TIME	Time
39	RLP	Resource Location Protocol
42	NAMESERVER	Host Name Server
43	NICNAME	Who Is
53	DOMAIN	Domain Name Server
67	BOOTPS	Bootstrap Protocol Server
68	BOOTPC	Bootstrap Protocol Client
69	TFTP	Trivial File Transfer
75		any private dial out service
77		any private RJE service
79	FINGER	Finger
95	SUPDUP	SUPDUP Protocol
101	HOSTNAME	NIC Host Name Server
102	ISO-TSAP	ISO-TSAP
113	AUTH	Authentication Service
118	UUCP-PATH	UUCP Path Service
123	NTP	Network Time Protocol
133-159		Unassigned
160-223		Reserved
224-241		Unassigned
247-255		Unassigned

表 3.1: 予約されている TCP ポート番号の例

Decimal	Keyword	Description
0		Reserved
1-4		Unassigned
5	RJE	Remote Job Entry
7	ECHO	Echo
9	DISCARD	Discard
11	USERS	Active Users
13	DAYTIME	Daytime
15	NETSTAT	Who is up or NETSTAT
17	QUOTE	Quote of the Day
19	CHARGEN	Character Generator
37	TIME	Time
39	RLP	Resource Location Protocol
42	NAMESERVER	Host Name Server
43	NICNAME	Who Is
53	DOMAIN	Domain Name Server
67	BOOTPS	Bootstrap Protocol Server
68	BOOTPC	Bootstrap Protocol Client
69	TFTP	Trivial File Transfer
75		any private dial out service
77		any private RJE service
79	FINGER	Finger
123	NTP	Network Time Protocol
133-159		Unassigned
160-223		Reserved
224-241		Unassigned
247-255		Unassigned

表 3.2: 予約されている UDP ポート番号の例

る。目的のソケット、つまり、サーバのソケットが存在して、コネクションを処理する準備をしているならば、コネクションは完成し、クライアントはサーバに要求を出し始めることができる。クライアントからのメッセージは、順序よくサーバに送られる。おのおののソケットが閉じられると、コネクションが切れる。コネクションが切れたあとにプロセスがメッセージを送ろうとすると、OS がそのプロセスに SIGPIPE シグナルを送る。そのプロセスは、そのシグナルを受信するとただちに終了する。

サーバの方は、ソケットを生成し、それに名前をバインドして、それからクライアントからの要求を受け付け始める。サンプルプログラムでは、ソケット番号を出力して、このソケットのために listen() を呼んでいる。いくつかのクライアントが同時にコネクト要求を出した場合、未処理のコネクションのための待ち列をシステムアドレス空間に維持しなければならない。listen() は、コネクションを待ち受けようとするソケットに印をつけ、待ち列を初期化する。コネクションが要求されると、それは待ち列に入れられる。待ち列がいっぱいになったら、エラー状態が要求者に返される。この待ち列の最大長は、listen() の 2 番目の引数で定義される。そして、その最大長は、システムによって限られている。一度 listen() が呼ばれると、サーバは無限ループに入る。ループを回るごとに、新しいコネクションが受け入れられ、待ち列から取り除かれて、そのコネクションのための新しいソケットが生成される。コネクションが生成されたら、サーバはフォークして子プロセスを作り、親プロセスはコネクションソケットを閉じて再びクライアントのアクセプトを行い、子プロセスはサービスを実行してコネクションソケットを閉じる。accept() は待ち列からの未処理のコネクト要求を受け付けるか、要求を待ちながらブロックする。メッセージは、コネクションソケットから読まれる。つながっているコネクションで read() を行うと、データを受信するまでブロックしている。read() で返される値は、読み込まれたデータのバイト数である。コネクションが切れると、リードコールが直ちに返される。そのときに返されるバイト数は 0 である。

3.3.2　両者の特徴と違い

V の方は、TCP を使用しているので、クライアントのソケットとサーバのソケットの間でコネクションが確立されている。従って、クライアントはまず最初に使用するサービス番号だけをサーバに送信し、その後からデータを送信する。１つのクライアントプロセスに対して１つのサーバプロセス（フォークされた子プロセス）が動作し、その間でコネクションが確立されているので、それぞれのクライアントが、各々に使用したいサービスを並列して同時に使うことができる。また、サーバは並列処理をして

いるから、１回の変換処理においてバッファサイズを越えるデータを扱うことができる。つまり、文字列終了文字を受信するまでは一連のデータであると判断し、それまでデータをためておくことができる。それから変換処理を行って、処理結果をバッファサイズごとに分割してクライアントに送信する。

T の方は、UDP を使用しているので、クライアントのソケットとサーバのソケットの間でコネクションが確立されていない。従って、クライアントは使用するサービス番号と変換するデータの組をサーバに送信する。サーバはシーケンシャル処理で受信データを扱っているので、１回の変換処理において受信用バッファサイズを越えるデータを扱うことはできない。クライアントがサービス番号と変換すべきデータをいっしょに送ろうとして、データが送信用バッファサイズを越えたとき、次に越えたデータだけを送ったとしても、サーバの方はそのデータがどのクライアントのどのデータに続くものなのかを判断することは困難である。よって、クライアントとサーバのそれぞれのバッファサイズの範囲内で１回の処理が行われる。

V の方は、クライアントとサーバの間にコネクションが確立されているので、データの送信において送り先のアドレスは必要ない。よって、プログラムでは、引数にアドレスを含まない write という関数を使用する。逆に T の方は、クライアントとサーバの間にコネクションは確立されていないので、送り先のアドレスが必要になる。よって、プログラムでは、引数にアドレスを含む sendto という関数を使用する。この場合、データを受信するときは送り元のアドレスを必要とするので、プログラムでは、引数にアドレスを含む recvfrom という関数を使用する。V の方は、データを受信するときに送り元のアドレスを必要としないので、プログラムでは、引数にアドレスを含まない read という関数を使用する。

V の方は、クライアント側が途中で使用するサービスを変更しようとするとき、それをサーバに知らせるためにサービス変更文字を送信する必要がある。両者とも、クライアントが１回のソケットの生成で何回も連続してサービスを使用できるようにしてあるので、外面的にはどちらも変わりはない。しかし、T の方は一回の実行ごとに選択するサービスの識別番号をクライアントからサーバに送らなければならないのに対し、V の方は最初にそれを送ればコネクションをつなげてあるので、あとはデータだけを送ればよいので、そのことだけを見れば T の方がサービス選択のエラーが生じやすい。しかし、T の方はコネクションをつなげていないので、サーバがクラッシュしたときにはそのリセットは V よりもやりやすい。つまり、T の方はリセットしたら切れたところから始められるが、V の方はまたコネクションをつなぎなおさなければならない。また、実行速度はほとんど

違いがない。

エラーリカバリーにおいては、V の方は使用サービスの決定とメッセージ交換が分かれていて、T の方は両方ともいっしょであるので、V の方が T よりもエラーの検出と修復がいくらかやりやすい。

サンプルプログラムでは、サーバは変換処理を行うときに 2 つのテンポラリファイルを使用するので、それが他のプロセスからアクセスされないようにしなければならない。そこで、このプログラムでは、両者ともテンポラリファイルを作るとき、プロセス ID をファイル名に使用して、その後で、そのファイルを消去している。V の方は、フォークして子プロセスにサービスを実行させるので、子プロセスが自分のプロセス ID を使ってテンポラリファイルを作る。つまり、親プロセスがフォークした分だけテンポラリファイルが用意される。

3.3.3　加えるべき制御機構

両者とも、ユーザがサーバのサービスを使用する前に、どのようなサービスがあって、現在動作しているサービスはどれなのかをメニューの形式で知らせる必要がある。これは、クライアントがサーバと最初の通信を行う前にユーザに知らせなければならない。現在動作しているサービスがどれなのかを自動的に調べるようなプログラムはないので、今のところは管理者が定期的に調べるしかない。サンプルプログラムでは、ユーザが使用するサービス番号とデータの入力方法をメニューから選択するようになっている。

また両者とも、クライアントの１回のソケットの生成で連続してサービスを使用できるようにしなければならない。サンプルプログラムでもわかるように、この機構はクライアントに依存されている。従ってサーバは、ただ到着したデータを受信して処理し、クライアントに送り返している。

3.3.4　問題点

提供されるサービスが正しいかどうかを調べることは非常に難しい。テストのためのファイルを用意してそれを使って確かめてみることもできるかもしれないが、この場合、このようなファイルを用意すればいいと決めることはかなり困難である。転送されるメッセージが正しいかどうかは、クライアント側にそれをエコーバックさせて知ることもできる。しかし実際のところ、このような単純な方法ではちょっと頼りないものがある。

適当なバッファの大きさを決定するのもかなり重要である。サンプルプログラムの T の方は、クライアントにおいては、送信用が２０４８で受信用が４０９６に指定してある。サーバにおいては、送信用も受信用も４０

９６に指定してある。サーバが処理した結果のバイト長が、クライアントが送信するデータのバイト長の二倍を越えることはないので、このような大きさに設定してある。クライアントのデータの送信において、一度に２０４７文字を越える文字列を扱うことができない。しかし V の方は、クライアントのデータの送信において、一度にバッファサイズを越える文字列を扱うことができるので、クライアントもサーバも２０４８に指定してある。バッファサイズを越えるデータを扱うときに、送信側の write を呼び出す回数と受信側の read を呼び出す回数が同じになるように、クライアントとサーバのバッファサイズを等しくしてある。さらに V も T も、入力ファイル名の文字数の合計が一度に２０４７を越えたら、２０４７文字目のところまでしか扱うことができない。２０４７文字を越えようとしたら、強制的に入力を終了する。しかし、入力ファイル名の方の問題は実際に起こる可能性は０に等しいので、考えなければならないのは T のバッファサイズの問題である。実際のサービス処理の能率を考えて、適当なバッファサイズを選択しなければならない。

3.3.5 サンプルプログラムにおける注意点

サーバプログラムを見てわかるように、ローマ字をひらがなに変換するために配列にひらがなの五十音を入れている。この配列に納められているひらがなを使って変換を行うときは、ソースプログラムを EUC モードにしてコンパイルを行わないと、実行可能のサーバプログラムが作成されない。よって、このサンプルプログラムのサーバとクライアントを使用するときは、画面を EUC モードにする必要がある。

サーバには、前で述べたように、起動するときに a というオプションをつけることができる。このオプションをつけると、どんなホストでも起動するが、つけないと、ホストによっては起動しないこともある。

コンソール入力で一度 ^D を使用すると、それから後のコンソール入力がブロックされてしまう。よって、入力の終了は、すべて「RET」.「RET」で行う。

配列に納められている「？＊@」の３文字の文字列と、コンソールから入力される「？＊@」の３文字の文字列では、コードが違うので、それぞれ区別される。よって、バーチャルサーキットにおいてクライアントが、サービス継続／変更文字である「？＊@」の３文字の文字列を、データメッセージとしてコンソールから入力してサーバに送信しても別に問題はない。

ローマ字からひらがなに変換するときは、英小文字によるローマ字だけが変換対象文字として判断され、英大文字などの他の文字は、そのまま出力される。

「ん」と母音が続くときは、間に x を入れる。

tani⟹ たに
tanxi⟹ たんい
tanka⟹ たんか

UNIX の規則において、１つのファイル名は２５５文字以内でスペースは含まないようにすることになっているので、これをもとにプログラムは作成されている。

第 4 章

広域分散型機械翻訳システムの設計

ネットワーク上に機械翻訳システムを構築するためには、Lisp で稼働する機械翻訳プロセスに、第３章で述べたプロセス間通信の考えを応用させなければならない。この章では、実際にそれを行っていくためのいくつかの方法を考察し、詳しく述べることにする。

4.1 ネットワークにおける自動翻訳機構の確立

ネットワーク上に広域分散型機械翻訳システムを構築するとき、それは大きく分けると次の３つのプロセスから成り立つ。

- 自動翻訳のためのクライアント（Automatic Translation Client : ATC ）
- スケジューリングサーバ（Scheduling Server : SS）
- 自動翻訳サーバ（Automatic Translation Server : ATS）

Lisp で稼働する機械翻訳システムは、本体だけで１０ Mbyte あるので、ATS は非常に重いプロセスになる。これだけの大きさになると、普通は１つの計算機内で翻訳プロセスをフォークすることはできないので、各計算機（各ホスト）には ATS を稼働させたとしても最大１つしか置くことができない。しかし、ネットワーク中には ATS が複数個存在することになる。よって、この ATS を管理するための SS が必要になる。SS は、ATC から ATS を使用する要求が来たら、複数の ATS の中から適切なところを使えるようにする。この場合、ATC と ATS の間は、直接コネクションをつなぐ方法と、SS を経由してつなぐ方法が考えられる。通信時間の短縮や SS の負荷軽減を考えると前者の方が望ましいが、セキュリティーの問題を考えると後者の方が望ましい。これの詳しいことは後で述べることにする。複数の ATS を管理する SS は、ネットワーク全体で１つだけでなければな

らない。ATC は翻訳を行うためのソーステキストを ATS に送信し、その結果を受信するためのプロセスである。ATC は SS に翻訳の要求を発行し、ATS とつながったらソーステキストを送信し、すべて結果を受信したら終了する。

4.1.1　自動翻訳のためのクライアント（ATC）

一般的にクライアントに求められる一番重要なことは、できる限り速く正確にサーバの処理結果をユーザに知らせることである。ATC を設計する際に、このことをまず第一に考え、セキュリティーの問題もあるかもしれないが、ATS と直接コネクションをつなぐようにした。また、きちんと通信が行われているかどうかを ATS と SS に把握させるために、一回のデータ受信ごとに確認応答（ACK）を返送する。逆に ATS や SS にデータを送信したときは、ACK を待つことになる。ACK の送受信を行うということは、その分だけ通信時間がかかることを意味するので、本当に信頼のできるプロセス間での通信を行うのであれば、これは必要のないものであるかもしれない。TCP を使用しているこのシステムでは、通信のエラーが発生する確率はほとんど 0 に等しい。しかし、万一のエラーに備えることはもちろん、Lisp で稼働する翻訳プロセスがいつ異常終了するかわからないので、この ACK の送受信は ATC を操作するユーザにとって必要なものである。人間の見た目には、この ACK の送受信における所要時間は、全く気にならない程度であるので、翻訳処理の遅延はほとんど起こらないに等しい。

ここでの翻訳処理は 4 つの行程に分かれている。順に、形態素解析、構文解析、構文変換、構文生成となる。ATS が順番にこれらを処理していく間に、ATC はそれぞれの結果を順番に受信していく。正常に終了したらその結果が、またエラーを伴って終了したらそのエラーを示すメッセージが受信される。正常な結果なのか、それともエラーを示す結果なのかは、ヘッダのコードで区別できる。この 4 つの行程のすべての結果をユーザに表示するか、または生成された最後の結果だけを表示するかは、ATC を起動するときに、オプションで決めることができるようにしてある。また、ATC は、最初に SS から接続要求認証を受け取った時点を 0 にして、処理結果や ACK を受信するごとにそこからの実行時間を計って表示するようにしてある。これは、4 つの行程でのおおまかな処理時間や通信時間がわかるので、非常に便利である。

4.1.2 スケジューリングサーバ（SS）

SS は、ATC からの要求を受信すると、起動している ATS を順番に検索していき、空いている ATS が見つかったら、それを ATC に割り当てる。つまり、その ATS のホスト名を ATC に送信する。そして、ATC からのホスト名認証を受信すると、SS は割り当てた ATS を使用中とし、その ATC とのコネクションを切って、再び別の ATC からの接続要求を待つ。あとは ATC と ATS の間での通信の進行状況を知らせるコードを ATS から受信し、SS はどこまで翻訳処理が進んでいるかを常に把握しておくようにする。そして、ATS から翻訳処理が終了したという知らせを受信したら、SS はその ATS を空きデーモンとする。

この一連の動作は、ATC と ATS が直接コネクションを確立する場合で、SS を経由してコネクションを確立する場合はちょっと違ってくる。このときは、SS は ATS からの処理結果を常に受信して、それから ATC にその結果を送信する。ACK の送受信も同じで、必ず SS を経由して ATC から ATS、あるいは、ATS から ATC に送られる。当然、一連の翻訳処理がすべて終了するまで、ATC と SS の間のコネクションは切られることはない。この 2 つの詳しいモデルは、次のセクションで述べることにする。

今のところでは、SS は ATC からの接続要求を受信すると、空いている ATS を探して、一番最初に見つかった空き ATS をその ATC に知らせる。もし、すべての ATS が使用中だったら、SS はその ATC に空いている ATS がないことを伝え、ATC がそれを認証したことがわかったらコネクションを切る。SS は、ATC の要求のための待ち列を用意していない。まだ ATS の使用率や平均使用時間などがはっきりしないためこのようにしてある。将来的にしっかりと広域分散型機械翻訳システムを確立するためには、このような機能は絶対に必要なものである。また、ATC が SS に接続要求を出すとき、ATC からは ATS を選択することはできない。これを行うためには、ATS があるきちんとした機能を持ち合わせていないと意味がなくなるので、これに関することは次の自動翻訳サーバのところで説明することにする。

4.1.3 自動翻訳サーバ（ATS）

ATS は、ATC から翻訳すべきデータを受信したら、それを翻訳して、その結果を ATC に送信する。前にも述べたように、翻訳処理には 4 つの行程があるので、それぞれの結果を順に ATC に送信する。このとき、ATS と ATC の間で直接コネクションが確立されていれば、例えば ATS が形態素解析を終了したら、その結果を直接 ATC に送信し、形態素解析が終了したという途中経過の状況だけを SS に知らせる。また、SS を経由してコ

ネクションが確立されているならば、ATS は形態素解析の結果をまず SS に送信し、SS が、通信を行っている ATC にそれを送信する。形態素解析、構文解析、構文変換、構文生成のどれをとっても実行された結果のデータはかなりの量なので、SS を経由する場合は、これにかかる負荷がかなり大きくなる。それを考えると、ATS と ATC は直接つながれている方が便利である。

本来 SS には、ATC を受け付ける窓口の役割と、ATS の４つの行程を管理する役割がある。１つの SS にこの２つの役割を任せてもいいが、ここでは、ATS の中に４つの行程を管理する SS を設けて、この SS と翻訳プロセスを合わせて ATS と呼ぶようにする。すなわち、おおもとの SS は、この ATS の中にある SS を管理することになる。現段階では、ATS が翻訳処理を実行するとき、Lisp 上で稼働する翻訳プロセスを呼び出してこのプロセスを立ち上げたら、ATS は双方向パイプによってデータの送受信を行い、翻訳処理の結果を得る。よって、この４つの行程は１つのプロセスになっているので、今のところ ATS の中にある SS は単なる ATS の窓口になっている。従って、１つの ATS には１つの ATC しか割り当てられない。複数の ATC を割り当てることもできなくはないが、４つの行程がすべて終了してから次の ATC の処理を行うので、これは行わない。これを意味あるものにするには、この４つの行程を独立に動くようにしなければならない。つまり、この４つの行程が４つのプロセスになることが重要である。このとき、ATS の中にある SS が威力を発揮するのである。例えば、ある ATC からの要求を形態素解析に割り当てているときに、別の ATC からの要求を構文解析に割り当てることもできる。すなわち、１つの ATS に複数の ATC を割り当てて意味をなすのである。このようになれば、SS は、ATC からどのような ATS につなげてほしいかを聞いて、処理時間の問題を考えた上で複数の ATC を１つの ATS につなげるようにすれば、希望をかなえることができる確率も高くなる。具体的には、ATS が select() という関数を使って、複数のソケットで ATC からの要求を待ち、おおもとの SS は ATC が求める条件に合った ATS をすべて検索し、利用できる ATS が見つかったら、そのホスト名を ATC に知らせる。１つの ATS に対して１つの ATC しかつなげないのであれば、希望を聞いても、偶然それに合う ATS が空いていなければ、利用することはできない。

このように４つの行程が独立に稼働していれば、かなり便利であることは言うまでもない。一連の流れでしか行えなければ、途中で結果が気に入らなくても最後までそのまま翻訳処理を実行していかなければならないが、４つの行程が４つのプロセスになっていると、例えば、構文解析の結果が気に入らないときに、それをユーザが手直しして、また構文解析を行

うこともできる。また、構文変換の結果と同じようなものをユーザが作成して、いきなり構文生成を行うことも可能である。

4.2 ネットワークにおける機械翻訳環境モデル

機械翻訳システムをネットワーク上に構築するためには、自動翻訳のためのクライアント（ATC）とスケジューリングサーバ（SS）と自動翻訳サーバ（ATS）が必要であることは前のセクションで述べた。この３つの要素からなる翻訳システムのモデルには以下の６つが考えられる。ここで、SS が２つ存在するモデルにおいては、ATS の中で翻訳処理における４つの行程を管理するスケジューリングサーバを SS2、このスケジューリングサーバを管理する本来の意味でのスケジューリングサーバを SS1 と呼ぶことにする。

1. ATC と SS2 が直接コネクションをはり、SS2 と ATS が双方向パイプでつながれているタイプ $\Longrightarrow$DCP 方式
2. ATC と SS2 の間で SS1 を経由して２本のコネクションをはり、SS2 と ATS が双方向パイプでつながれているタイプ $\Longrightarrow$IDCP 方式
3. ATC と SS2 が直接コネクションをはり、SS2 と ATS の４つの行程がそれぞれ４本のコネクションでつながれているタイプ $\Longrightarrow$DCC 方式
4. ATC と SS2 の間で SS1 を経由して２本のコネクションをはり、SS2 と ATS の４つの行程がそれぞれ４本のコネクションでつながれているタイプ $\Longrightarrow$IDCC 方式
5. ATC を管理する SS がフォークして、子プロセスの SS と ATS が双方向パイプでつながれているタイプ $\Longrightarrow$FP 方式
6. ATC を管理する SS が同時に ATS の４つの行程を管理しているタイプ $\Longrightarrow$CC 方式

4.2.1 DCP 方式

DCP 方式（Direct Connection and Pipe、Figure4.1参照）は、次のような通信形態をとる。

1. ATC が SS1 に空いている SS2（ATS）を使用したいという要求を出す
2. SS1 は空いている SS2（ATS）を探し、そのホスト名を ATC に知らせる

3. ATC はそのホストの SS2 と直接通信を開始する

4. SS2 は ATS を起動して、ATS は SS2 と双方向パイプでつながる

5. SS2 は ATS の４つの行程のうちの１つが終了するごとに、ATS からの翻訳結果を直接 ATC に送信した後、SS1 に途中経過を知らせる

6. SS2 は翻訳処理がすべて終了したら、そのことを ATC に知らせて、その後 SS1 に知らせる

この方式では、前にも述べたように SS2 と ATS は双方向パイプでつながれているので、SS2 は単に ATS の窓口の役割しか果たしていない。ATC から通信開始要求が到着すると、SS2 は Lisp プロセスを立ち上げて、そこで翻訳システムを起動する。これが ATS となる。このとき、この翻訳プロセス（ATS）と SS2 の間で、双方向パイプが結ばれる。それから SS2 は ATC から翻訳を実行したいソースデータを受信し、これを一文ずつ区切って ATS に送信して翻訳を行う。この ATS は、形態素解析、構文解析、構文変換、及び構文生成の４つの行程を一連の処理として扱うので、途中の処理結果を ATC に返すことと、エラーが生じたときにその一連の流れを止めること以外のことを行うことは許されない。ATS が構文解析を行ってから、その処理結果を ATC が手直ししてまた ATS に送り返して、再度 ATS が構文解析を行うというようなことはできない。ATS は、構文解析を行ってその結果を得ると、それをそのまま SS2 に送信するわけではないからである。つまり、ATS は Lisp としての結果（次に構文変換を実行できる結果）は保持しておいて、ユーザにわかりやすいような結果になおしてそれをＣの文字列にして SS2 に送信している。SS2 はそれを ATC に送信しているので、ユーザが見た結果を手直ししても無駄なのである。Lisp の結果をそのままユーザに送り返すことは、システムの都合上ちょっと無理なことである。

SS1 が空いている SS2（ATS）を探してすべて使用中であったら、SS1 は ATC にすべての SS2（ATS）が使用中であることを伝えて、お互いにソケットを閉じて、コネクションを終了する。今のところは、待ち列を用意していない。

SS1 は、ATC に空いている SS2（ATS）のホスト名あるいはすべて使用中であるということを知らせたら、ATC とのコネクションを切る。SS1 は、それ以後の ATC の状態を、SS2 が送信してくる ACK によって判断することになる。この形態では、SS2 の責任はかなり重くなるが、SS1 は、複数の ATC からの要求と、SS2 からの ACK を処理しなければならないので、少しでも SS1 の負担を軽くすることを考えるとこの形態が一番いいよ

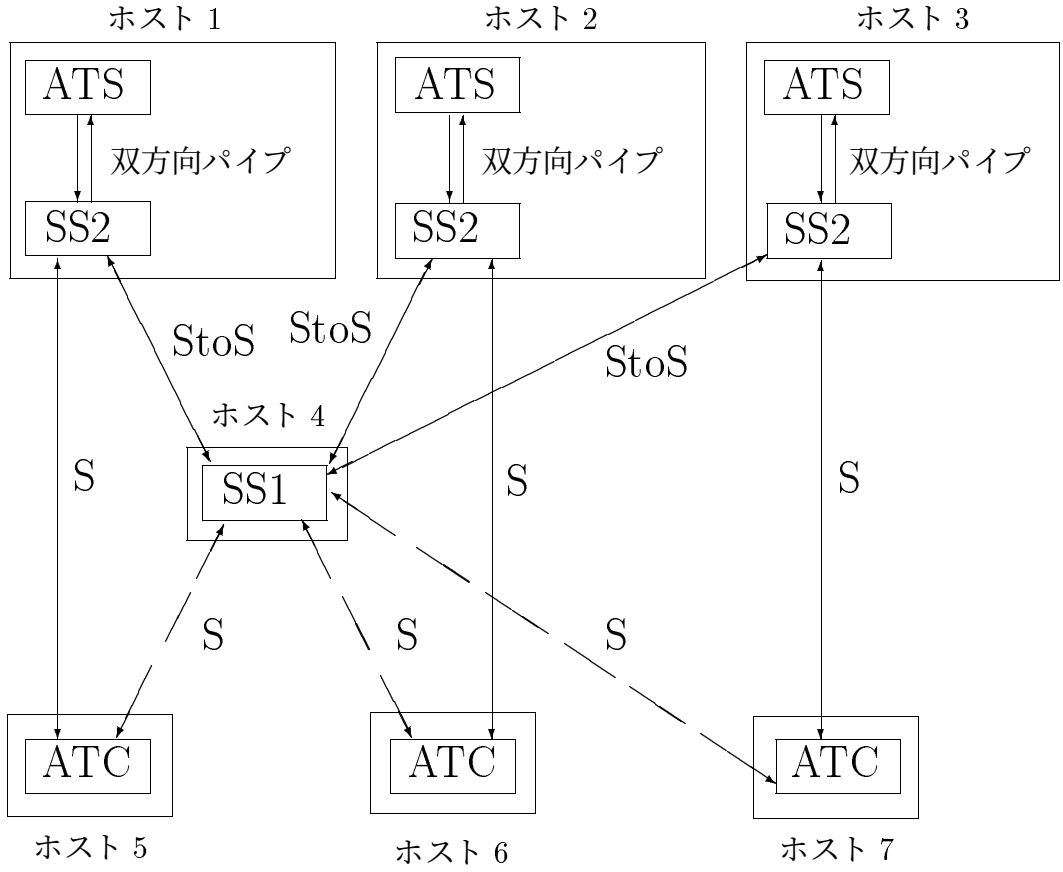
ホスト 1
ホスト 2
ホスト 3
ATS
ATS
ATS
双方向パイプ
双方向パイプ
双方向パイプ
SS2
SS2
SS2
StoS
StoS
StoS
ホスト 4
SS1
S
S
S
S
S
S
ATC
ATC
ATC
ホスト 5
ホスト 6
ホスト 7

図 4.1: DCP 方式

うに思われる。SS2 から SS1 に途中経過の ACK を一切送らないで、最後に翻訳が終了したということだけを伝えるという方法も考えられるが、少しでも正確な通信を行うことと、途中で ATC か SS2 か ATS のどれかがダウンしてしまったことをいち早く SS1 が知るためには、途中経過の ACK の送受信は行った方が望ましい。SS1 がダウンしてしまったら、どうしようもないことは言うまでもない。SS1 の処理速度によって、途中経過の ACK による翻訳処理の遅延はかなり変わるので、これをしっかりさせることは重要である。現段階で SS1 は、ATC からの受け付けと複数の SS2 との通信を待つことを１秒ごとに切り変えているので、この時間をきちんと決める必要がある。これからの経験によって、この値もしくは ACK の送受信の必要性も変わってくると思われる。

4.2.2　IDCP 方式

IDCP 方式（InDirect Connection and Pipe、Figure4.2参照）は、次のような通信形態をとる。

1. ATC が SS1 に空いている SS2（ATS）を使用したいという要求を出す
2. SS1 は空いている SS2（ATS）を探し、SS1 がその SS2（ATS）にコネクションをはる
3. ATC は SS1 を経由して、その SS2 と通信を行う
4. SS2 は ATS を起動して、ATS は SS2 と双方向パイプでつながる
5. SS2 は ATS の４つの行程のうちの１つが終了するごとに、ATS からの翻訳結果を SS1 に送信した後、SS1 が ATC に受信した結果を送信する
6. SS2 は翻訳処理がすべて終了したら、そのことを SS1 に知らせて、SS2 と SS1 の間のコネクションを切り、その後 SS1 が ATC にそのことを知らせて、SS1 と ATC の間のコネクションを切る

この方式も、基本的には DCP 方式と変わらない。SS2 と ATS が双方向パイプでつながれていることと、SS1 はすべての ATS が使用中であったらそのことを ATC に伝えて、SS1 と ATC の間のコネクションを終了してしまうことは DCP 方式と同じである。最も違うところは、翻訳処理の結果はもちろん通信での ACK から何からすべて、SS2 から ATC あるいは ATC から SS2 へ送信するとき、必ず SS1 を経由することである。

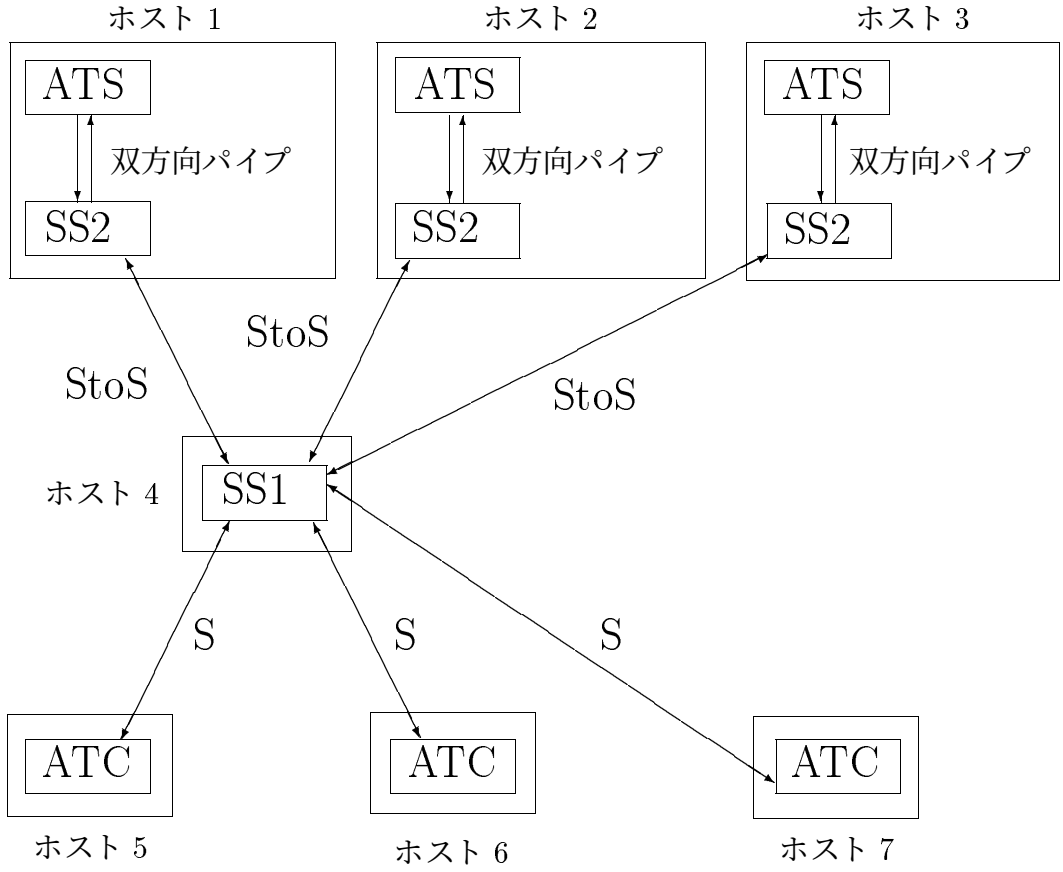
ホスト 1
ホスト 2
ホスト 3
ATS
双方向パイプ
SS2
ATS
双方向パイプ
SS2
ATS
双方向パイプ
SS2
StoS
StoS
StoS
ホスト 4
SS1
S
S
S
ATC
ATC
ATC
ホスト 5
ホスト 6
ホスト 7

図 4.2: IDCP 方式

SS1 が ATC からの接続要求を受信すると、DCP 方式では空いている SS2 のホスト名を ATC に返すが、この方式では SS1 自身が空いている SS2 にコネクションを張りにいく。つまり、ユーザ（ATC）に SS2（ATS）のことを知られなくてすむ。ATC は実際には SS1 とだけ通信を行っているが、あたかも ATS と通信を行っているかのように見える。DCP 方式では、ATC と SS2 は直接通信を行うので、SS2 あるいは ATS のことが見えてしまう恐れがある。よって、セキュリティーが問題になってくる。また、ATC が直接 SS2 と通信を行うときは、ATC、SS、ATS といったすべてのプロセスが動いているネットワーク内では、同種のネットワークアドレスを使用しなければならない。しかし IDCP 方式では、ATC には SS1 のことしかわからないし、ATC が理解していなければならないネットワークアドレスは SS1 のアドレスだけである。SS1 と SS2（ATS）の間で使われているアドレスを ATC が知っている必要はない。

IDCP 方式で最も問題になることは、実行速度と SS1 の負荷のことである。ATC と SS2 （ATS）を直接つないでいる DCP 方式は、SS1 を経由するこの方式よりは実行速度が速い。また、必要以上に SS1 にデータを送信しないので、SS1 の負荷も比較的軽くてすむ。ところが IDCP 方式では、ATS が翻訳して得た莫大なデータから通信の ACK まですべてが SS1 を経由する。しかも、１つの SS1 が複数個の SS2（ATS）と ATC の通信のフィルタを行わなければならない。これだけでも実行速度や SS1 の負荷にかなり影響している。

DCP 方式では、ATC が SS1 から空いている SS2（ATS）のホスト名を受信すると、ATC と SS1 の間でのコネクションを切る。よって、SS1 が実質的に通信のために使用しなければならないソケットの数は、SS2 につなぐ分と ATC からの要求を待つためのもの、つまり、SS2 の数＋１となる。しかし IDCP 方式では、一連の翻訳処理がすべて終了するまでは ATC と SS1 の間でのコネクションはつないだ状態でなければならないので、SS1 が通信のために使用しなければならないソケットの数は、SS2 につなぐ分と ATC につなぐ分、そして新たな ATC からの要求を待つためのもの、つまり、SS2 の数＋ ATC の数＋１となる。ATC の数が増えるにつれて、両者の SS1 にかかる負荷の差はどんどん広がっていく。また、SS1 がダウンしたときの影響もこの方式の方が大きい。

4.2.3　DCC 方式

DCC 方式（Direct Connection and Connection、Figure4.3参照）は、次のような通信形態をとる。

1. ATC が SS1 に条件付きの ATS 使用要求を出す

2. SS1 はその条件に合った ATS を管理する SS2 を順に検索し、接続可能になっている SS2 のホスト名を ATC に知らせる

3. ATC はそのホストの SS2 と直接通信を開始する

4. SS2 は ATC から送られてきたソースデータの内容にしたがって、ATS として独立に動く４つの行程のうちの１つを実行する

5. SS2 は４つの行程のうちの１つが終了するごとに、その翻訳の結果を直接 ATC に送信した後、SS1 に途中経過を知らせる

6. SS2 は ATC からの結果の ACK によって、次の動作を決める

7. ATC からの結果の ACK が翻訳処理終了であったら、SS2 は ATC とのコネクションを切り、そのことを SS1 に知らせる

この方式は、DCP 方式の機能を拡張したモデルである。DCP 方式では、SS2 と ATS が双方向パイプでつながれているので、１つの ATS には１つの ATC しか割り当てられないが、DCC 方式は、SS2 と ATS の４つの行程がそれぞれ４本のコネクションで結ばれているので、うまくスケジュールさせれば１つの ATS に一度に複数の ATC を割り当てることができる。ここでの割り当てるという意味は、待ち列に入れておくことではなく、ATC が処理を行っている状態を指している。

ATC はまず SS1 に条件付きの翻訳プロセス使用要求を送る。この条件というものは、どのような ATS を使用したいかということを示したコードである。例えば、どのような分野の翻訳を行いたいとか、できる限り速い翻訳システムを使用したいとか、このようなマシン上のシステムを使用したいといったことが挙げられる。その条件に合わせて、SS1 は SS2 を検索するのである。SS2 は独立に動く４つの行程で１つの翻訳システムを形成する ATS を管理するスケジューラである。この SS2 はそれ自身が管理する ATS の能力に合ったそれぞれの待ち列を持ち、最大いくつまでの ATC を一度に処理するかを決める。４本のコネクションを使用して４つの行程を管理しているので、理論上は４つの ATC までは一度に接続することは可能である。ATS が稼働するホストと SS2 が稼働するホストは、必ずしも同じでなければならないということはないが、Lisp で動く翻訳システムはダウンする率が決して低くはないので、同一ホストにある方が便利であるためこの方式を用いている。

SS1 はそれぞれの ATS の実行状態を把握しているわけではない。それを把握しているのは SS2 である。従って SS1 と SS2 という２つのスケジューリングサーバが存在するのである。負荷を分散させることが第一の

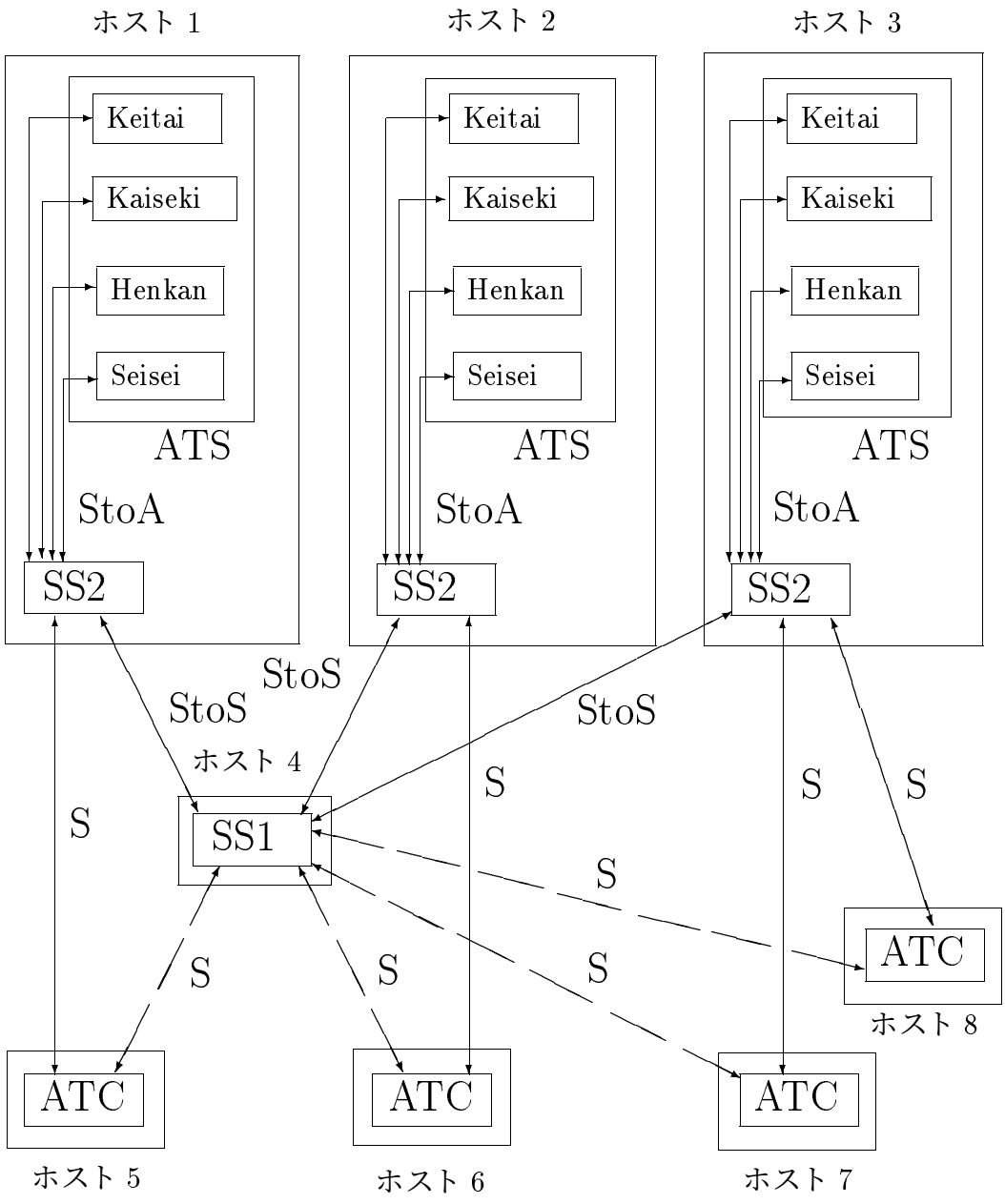
ホスト 1
ホスト 2
ホスト 3
Keitai
Kaiseki
Henkan
Seisei
ATS
StoA
SS2
StoS
StoS
StoS
ホスト 4
S
SS1
S
S
S
S
S
S
ATC
ホスト 8
ATC
ATC
ATC
ホスト 5
ホスト 6
ホスト 7

図 4.3: DCC 方式

目的である。SS1 は、SS2 の ATC 割り当て状態（どの行程を使用しているか）と待ち列待機状態（どの行程から開始する ATC が待っているか）を見て、接続可能かどうかを判断する。SS1 はあらかじめ ATC からどの行程から始めたいかを送信させ、それをもとに判断する。ATC の要求に合った ATS を管理する SS2 の１つが接続可能であれば、SS1 はそのホスト名を ATC に伝えて、ATC はその SS2 と直接通信を行う。接続可能の SS2 がなかったら、SS1 の待ち列に入れておいて一定時間待ってからまた検索してそれでもなかったらそのことを ATC に知らせてコネクションを切る。

DCP 方式と DCC 方式の最大の違いは、ATS の４つの行程が独立に動くか否かである。DCP 方式は、４つの行程が一連のプロセスになっているので、エラーが起こらない限りは形態素解析から始まって構文生成で終わるまでを ATS は順序よく必ず実行する。しかし DCC 方式は、どの行程から始まるか、いくつの行程を実行するか、処理結果を手直しして同じ行程を繰り返すのかなどがすべて ATC の希望によって決まるのである。ATC は始めにソースデータを送信するときに、いっしょにどの行程から始めたいかを SS2 に知らせる。デフォルトは形態素解析からである。１つの行程の処理が終わったら、SS2 は ATC に直接その結果を送信する。この結果を見てユーザ（ATC）は、そのまま終了するのか、手直ししてもう一度繰り返すのか、次の行程に進むのかを ACK とともに SS2 に知らせる。当然繰り返すときは、手直ししたデータを SS2 に送り返す。ATC は納得のいった時点で SS2 に翻訳処理終了のコードを送り、SS2 がそれを受信したら、両者の間のコネクションを切って、このことを SS2 が SS1 に知らせて一連の流れが終了する。

4.2.4 IDCC 方式

IDCC 方式（InDirect Connection and Connection、Figure4.4参照）は、次のような通信形態をとる。

1. ATC が SS1 に条件付きの ATS 使用要求を出す
2. SS1 はその条件に合った ATS を管理する SS2 を順に検索し、SS1 が接続可能になっている SS2 にコネクションをはる
3. ATC は SS1 を経由して、その SS2 と通信を行う
4. SS2 は SS1 を経由して ATC から送られてきたソースデータの内容にしたがって、ATS として独立に動く４つの行程のうちの１つを実行する

5. SS2 は４つの行程のうちの１つが終了するごとに、その翻訳の結果を SS1 に送信した後、SS1 が ATC に受信した結果を送信する

6. SS2 は SS1 を通ってきた ATC からの結果の ACK によって、次の動作を決める

7. ATC からの結果の ACK が翻訳処理終了であったら、SS1 は ATC とのコネクションを切り、そのことを SS2 に知らせて、SS1 と SS2 の間のコネクションを切る

この方式は、IDCP 方式の機能を拡張したモデルである。よって、IDCP 方式に比べて機能が拡張したところは、ATC と SS2 の通信の間に必ず SS1 が入っているという点を除けば、DCP 方式を DCC 方式に機能拡張した部分と同じである。また、DCC 方式と IDCC 方式の違いも、DCP 方式と IDCP 方式の違いと同じである。しかし、ここで特に考えなければならないことは、やはり SS1 にかかる負荷のことである。

IDCP 方式は、SS2 と ATS が双方向パイプでつながっているため、SS1 と SS2 の間にはられるコネクションの数は多くて起動している SS2（ATS）の数だけである。ところが、IDCC 方式では、SS2 は複数の ATC からの要求を一度に引き受けることができるので、その分だけ SS1 と SS2 の間にコネクションがはられることになる。仮に、起動している SS2（いっしょに ATS も起動していると仮定する）の数が３つであるとする。このとき、理論上は１つの SS2 に４つの ATC からの要求を一度に割り当てることができるから、SS1 と SS2 の間には最大４ × ３＝１２本のコネクションが結ばれることになる。そのとき SS1 と結ばれている ATC の数も１２個だから、SS1 は最も多くのソケットを処理しているときには、その数が２４個になってしまう。それの他にもう１つ新たな ATC に対するソケットを用意しなければならない。これが単なる ACK の送受信だけを扱うならまだしも、莫大な大きさの翻訳結果を中継しなければならないのであるから、SS1 にかかる負荷は、IDCP 方式には比べものにならないほどの大きさである。これに加えて SS1 は、ATC に対する待ち列の処理や、SS2 の状態管理を行わなければならないので、この方式では、SS1 がかなり効率良くかつ高速に動けるようなマシン上に存在しないと、システム全体の遅延は免れない。

4.2.5　FP 方式

FP 方式（Fork and Pipe、Figure4.5参照）は、次のような通信形態をとる。

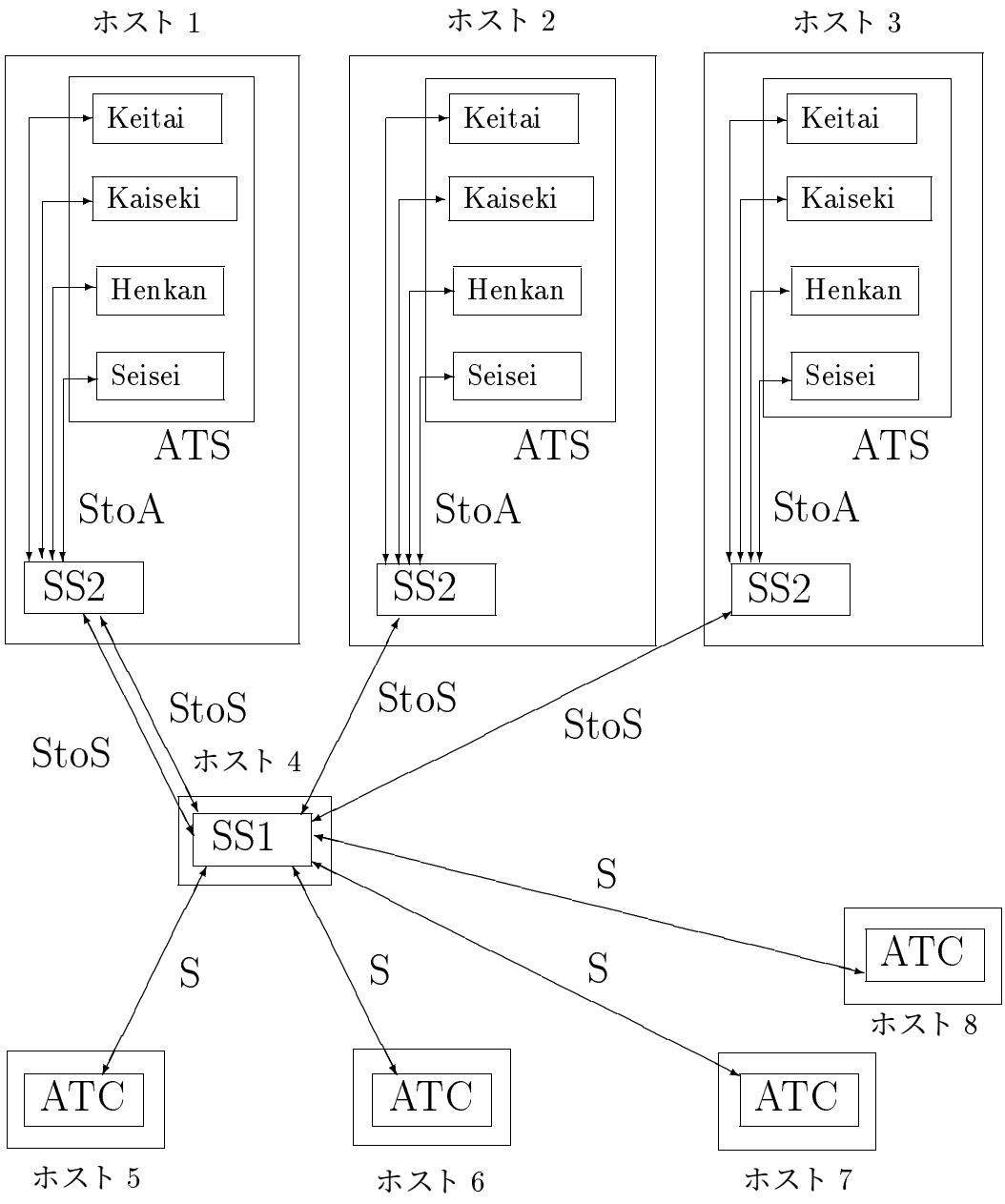
ホスト 1
ホスト 2
ホスト 3
Keitai
Kaiseki
Henkan
Seisei
ATS
StoA
SS2
StoS
ホスト 4
SS1
S
ATC
ホスト 5
ホスト 6
ホスト 7
ホスト 8

図 4.4: IDCC 方式

1. ATC が SS に ATS を使用したいという要求を出す
2. SS はフォークして子プロセスを作る
3. 親プロセスは、新たな ATC からの接続要求を待つ
4. ATC は子プロセスの SS と通信を開始する
5. 子プロセスの SS は、ATS を起動して、ATS はその SS と双方向パイプでつながる
6. 子プロセスの SS は、ATS の４つの行程のうちの１つが終了するごとに、ATS からの翻訳結果を ATC に送信する
7. 子プロセスの SS は、ATC に構文生成の結果（途中でエラーが生じたらそのときの結果）を送信したら、プロセスを終了しコネクションを切る

この方式は、形式的には第３章で述べたサンプルプログラムのクライアント／サーバモデルにおけるバーチャルサーキット型と同じである。SS は ATC からの接続要求を受信すると、フォークして子プロセスを作る。このとき、子プロセスは ATC との翻訳通信を開始し、親プロセスは新たな ATC からの接続要求を待機する。

FP 方式の利点は、アルゴリズムが簡単なことである。親プロセスの SS に対して子プロセスの SS は、他の方式で言えば SS1 に対する SS2 である。他の方式での SS1 と SS2 は完全に別のプロセスでしかもそれぞれ別のホストで稼働しているので、本来のスケジューラの役割を果たす SS1 は常に SS2 の動きを把握しておく必要性がある。しかし、この方式での親プロセスの SS と子プロセスの SS は別のプロセスではあるが、同一ホストに存在し、子プロセスは翻訳処理が終了したらホストのメモリ上からは消えてしまうので、親プロセスは子プロセスがゾンビになってマシンに残らないようにすること以外は、別に子プロセスの途中経過を常に把握しておく必要性はそれほどない。従って、アルゴリズムが簡単なのである。

しかし、FP 方式には最大の難点がある。それは、この複数の子プロセスのフォークに耐えうるだけのハードウェアが用意できるかどうかということである。莫大な金を投資すれば用意できるかもしれないが、それでは話にならない。１つの翻訳システム（ATS）は、約１０ M の記憶容量を必要とする。SS が３つの ATC と通信を行おうとすると、１つのマシン上に３つのフォークされた SS がそれぞれに ATS を立ち上げるので、このとき起動される３つの ATS だけで３０ M の記憶容量を使うことになる。もし、

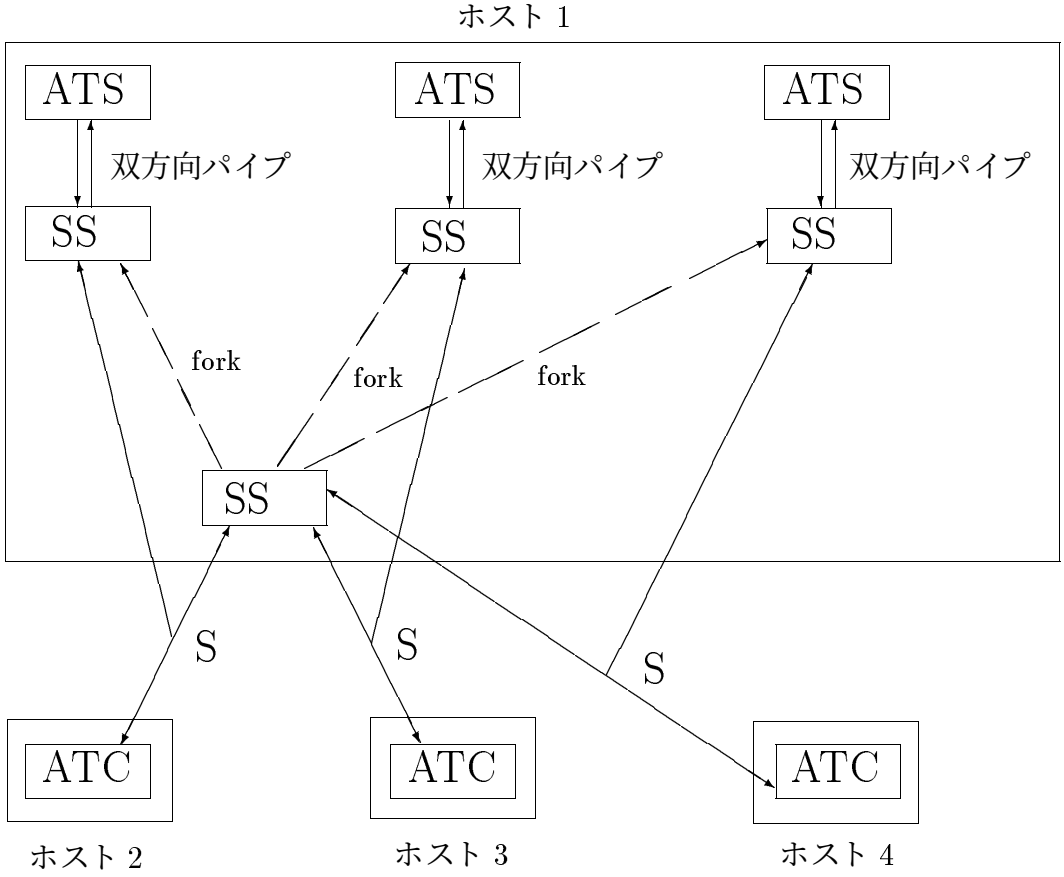
ホスト 1
ATS
ATS
ATS
双方向パイプ
双方向パイプ
双方向パイプ
SS
SS
SS
fork
fork
fork
SS
S
S
S
ATC
ATC
ATC
ホスト 2
ホスト 3
ホスト 4

図 4.5: FP 方式

これが実現できるマシンが無理をせずに手に入るならば、この方式を採用することが広域分散型機械翻訳システムを構築させる一番の近道である。

4.2.6　CC 方式

CC 方式（Connection and Connection、Figure4.6参照）は、次のような通信形態をとる。

1. ATC が SS に ATS のどの行程を使用したいという要求を出す
2. SS はその行程のプロセスへのソケットが空いていればその ATC と通信を開始する
3. SS はその行程のプロセスから処理結果を得たら、それを ATC に送信する
4. ATC は次に何を実行してほしいかを SS に知らせる
5. SS はその要求に従って実行する
6. SS は ATC から翻訳処理終了の要求を得たら、ATC とのコネクションを切る

この方式は、IDCC 方式の変型版である。IDCC 方式は、４つの行程のプロセスが同一ホストで動いているが、この方式はそれぞれの行程のプロセスが違うホストで動いている。これの一番の利点は、１つのホストにかかる負荷が軽減されることである。またホストによって動いている行程が決まっているので、SS が管理しやすい。

図では１つのホストに１つのある行程のプロセスしか動いていないが、これは複数個動いていても構わない。この場合は、他の方式のようにこの複数個動くある行程のプロセスのための SS が必要になる。こうなるとほとんど IDCC 方式と変わらなくなる。ホストごとに動いている行程が決まっているとはいえ、やはり SS にかかる負荷は相当なものである。

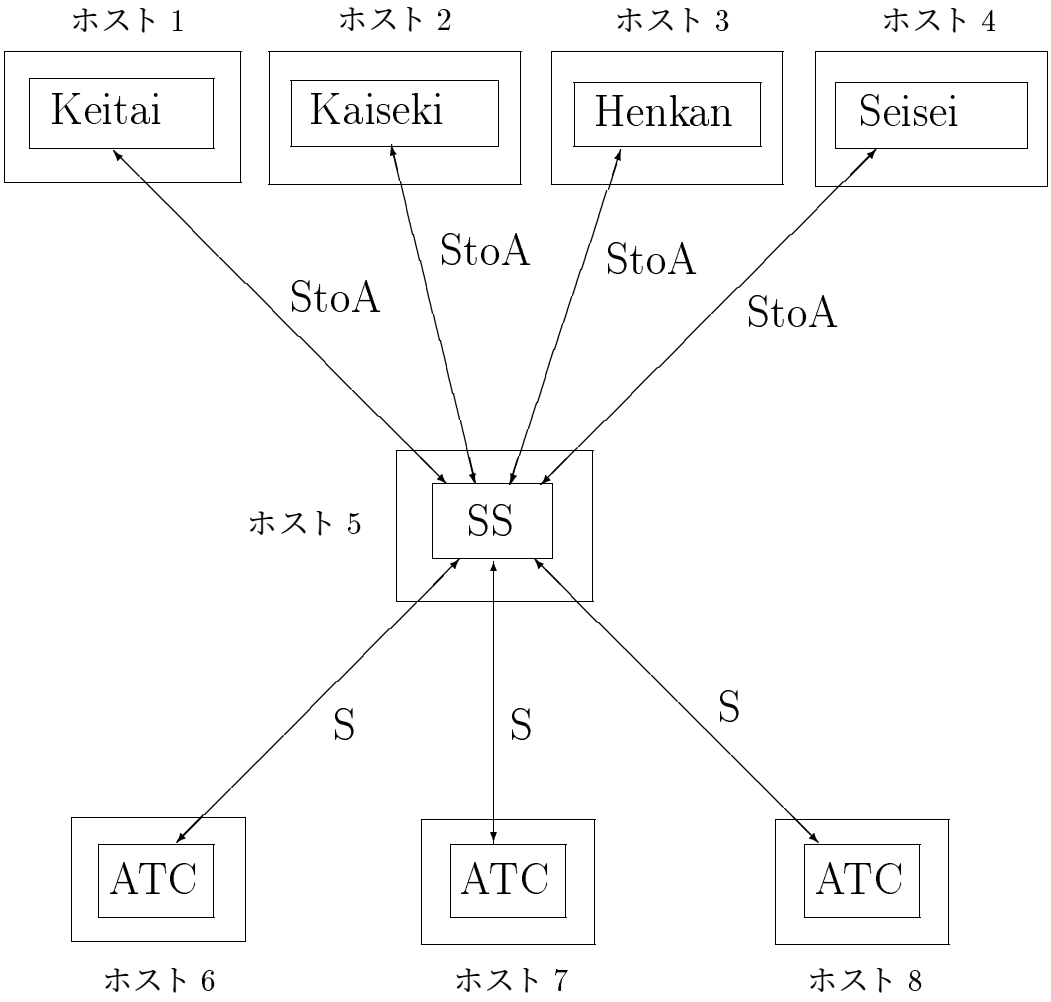
ホスト 1
ホスト 2
ホスト 3
ホスト 4
Keitai
Kaiseki
Henkan
Seisei
StoA
StoA
StoA
StoA
ホスト 5
SS
S
S
S
ATC
ATC
ATC
ホスト 6
ホスト 7
ホスト 8

図 4.6: CC 方式

第 5 章

インプリメンテーション

前の章で、６つの翻訳環境モデルについて述べた。それぞれに利点があるが、ここでは、その中で一番最初に述べた DCP 方式による広域分散型機械翻訳システムをネットワーク上に実装することにする（付録２参照)。既存の機械翻訳システムの４つの行程が独立に稼働するようになればこれを DCC 方式に拡張することもできる。また、通信時間の短縮と、スケジューリングサーバの負荷の軽減を第一の目的としたため、IDCP 方式ではなく、DCP 方式を採用した。

5.1 通信の流れ

ネットワーク上に機械翻訳システムを構築させれば、当然そのシステムの動作には一連の通信の流れができあがる（Figure5.1、Figure5.2、Figure5.3、Figure5.4、Figure5.5参照）。このセクションでは、DCP 方式の通信の流れを追うことによって、どのように機械翻訳が行われているかを述べることにする。

DCP 方式は、まず SS1 が起動するところからシステムの動作が始まる。起動するべきホストはデフォルトで決まっているが、別のホストで起動することもできる。しかしその場合は、SS2 や ATC が接続要求を出すときに、SS1 が起動しているホストをオプションで示さなければならないので、特に何もなければ、SS1 はデフォルトで決められたホストで起動するべきである。この SS1 は、広域分散型機械翻訳システムの中核を成すスケジューラである。これがダウンすると、システム全体がダウンすることになる。SS1 は起動すると、その時点から接続要求を受け付け始める。接続要求を受け付けるためのソケットと、デーモン（SS2）と通信を行うためのソケット（システム全体で起動することが許されているデーモンの個数だけ存在）を決められたタイムアウトごとに順に調べていき、要求が到着していたら、処理を行ってまた次のソケットを調べに行く。つまり、SS1 はタイムシェアリングにサービスを提供している。従って、ATC や SS2 はあ

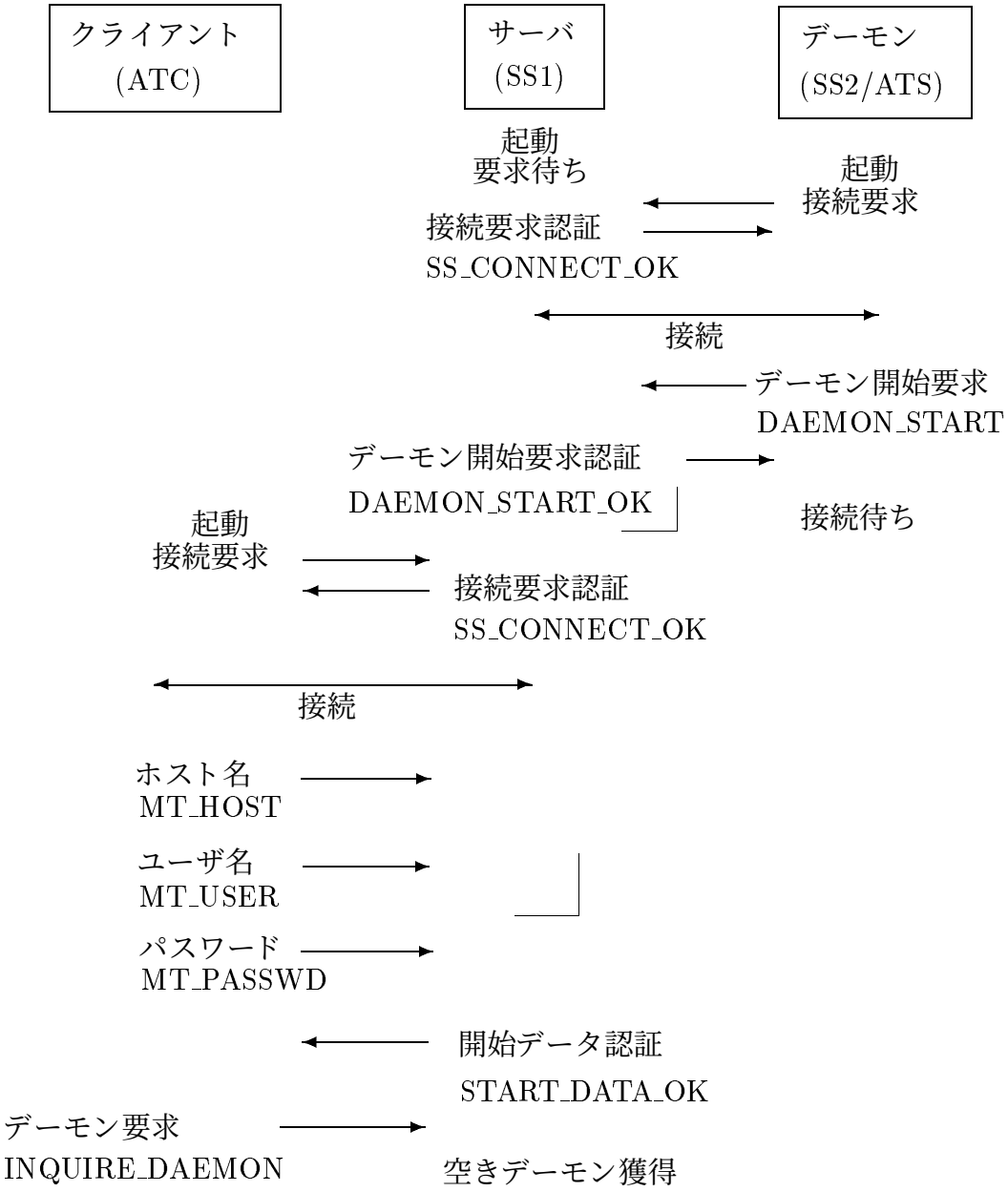

図 5.1: ステートダイアグラム 1

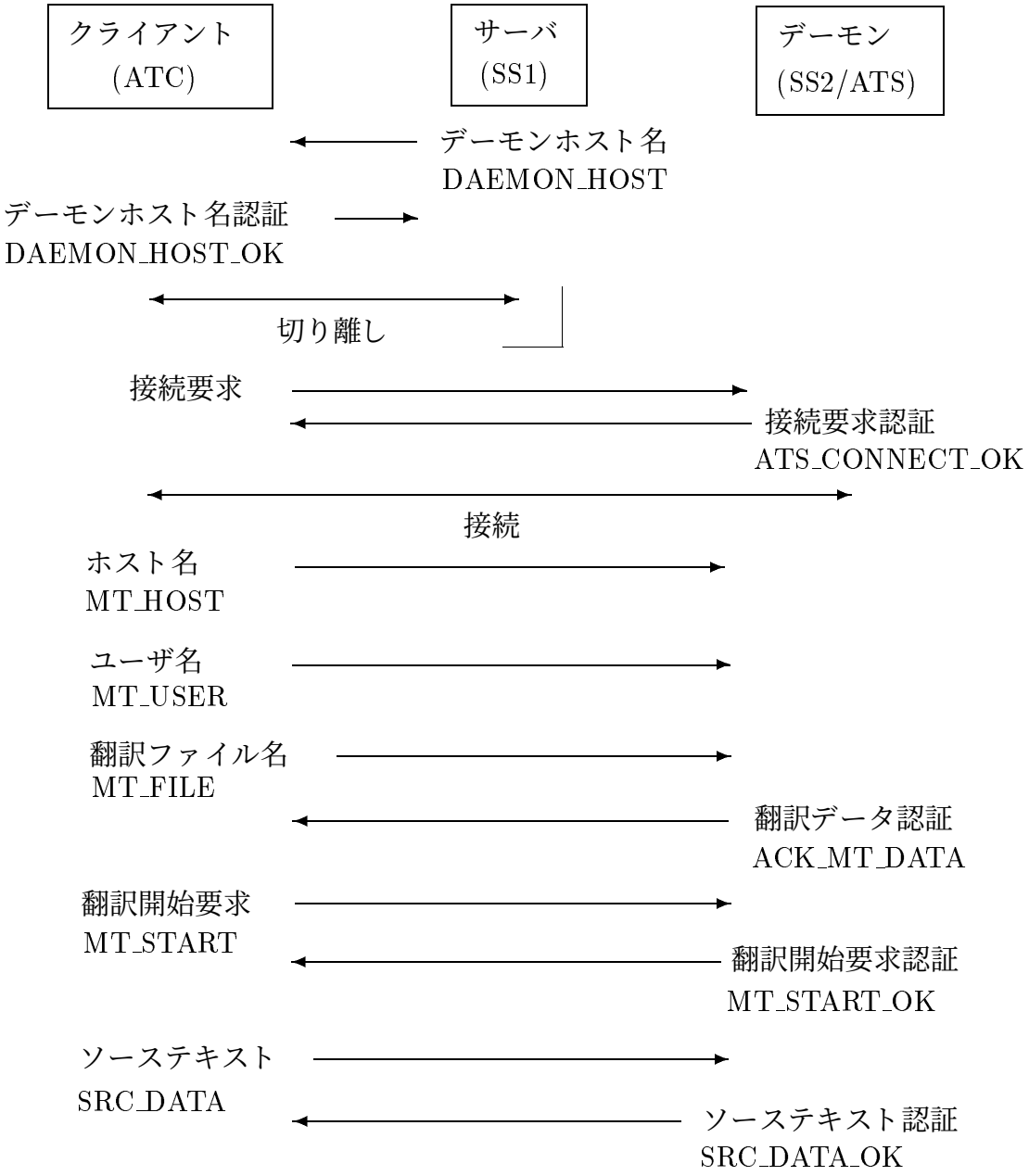

図 5.2: ステートダイアグラム 2

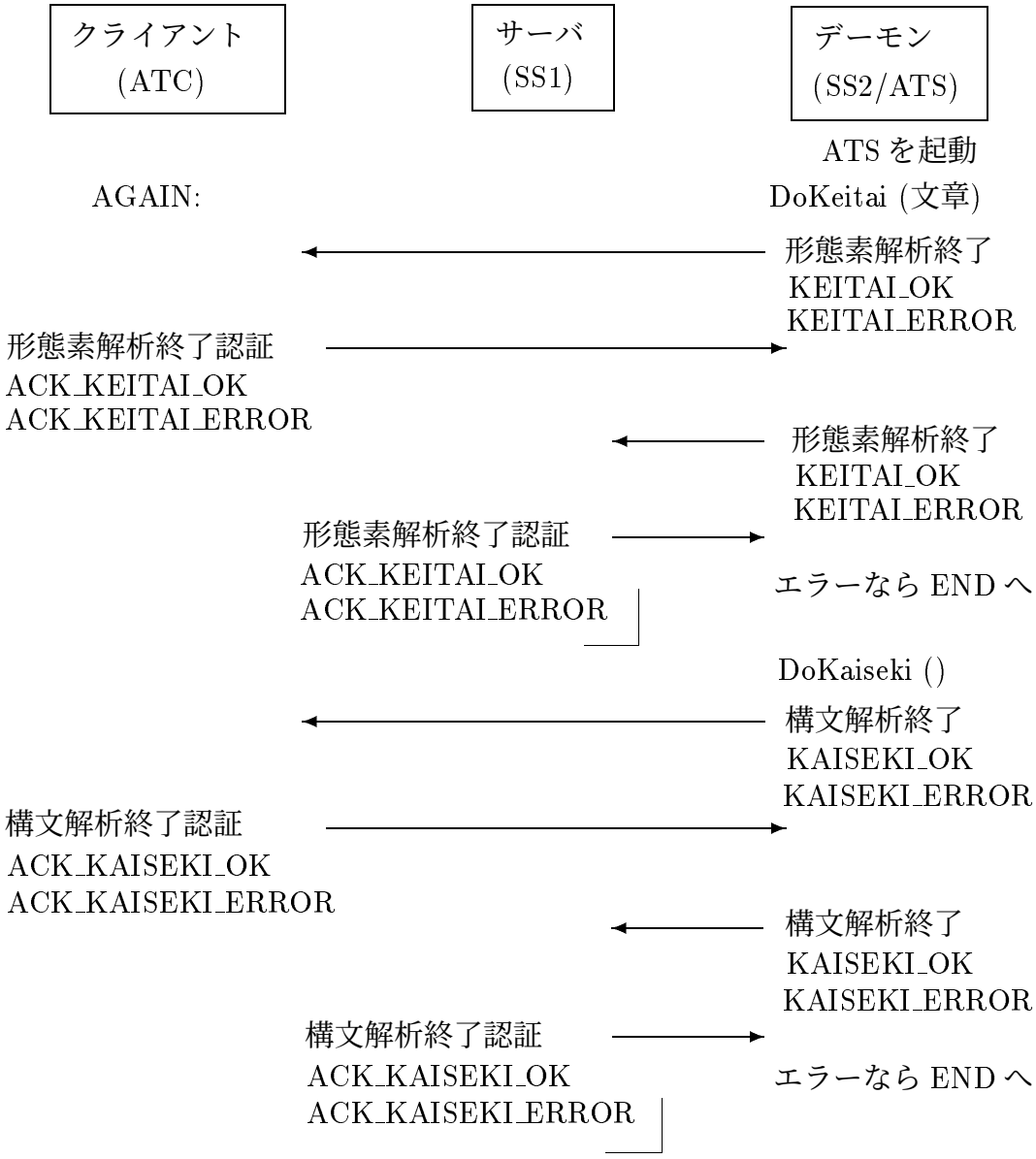

図 5.3: ステートダイアグラム 3

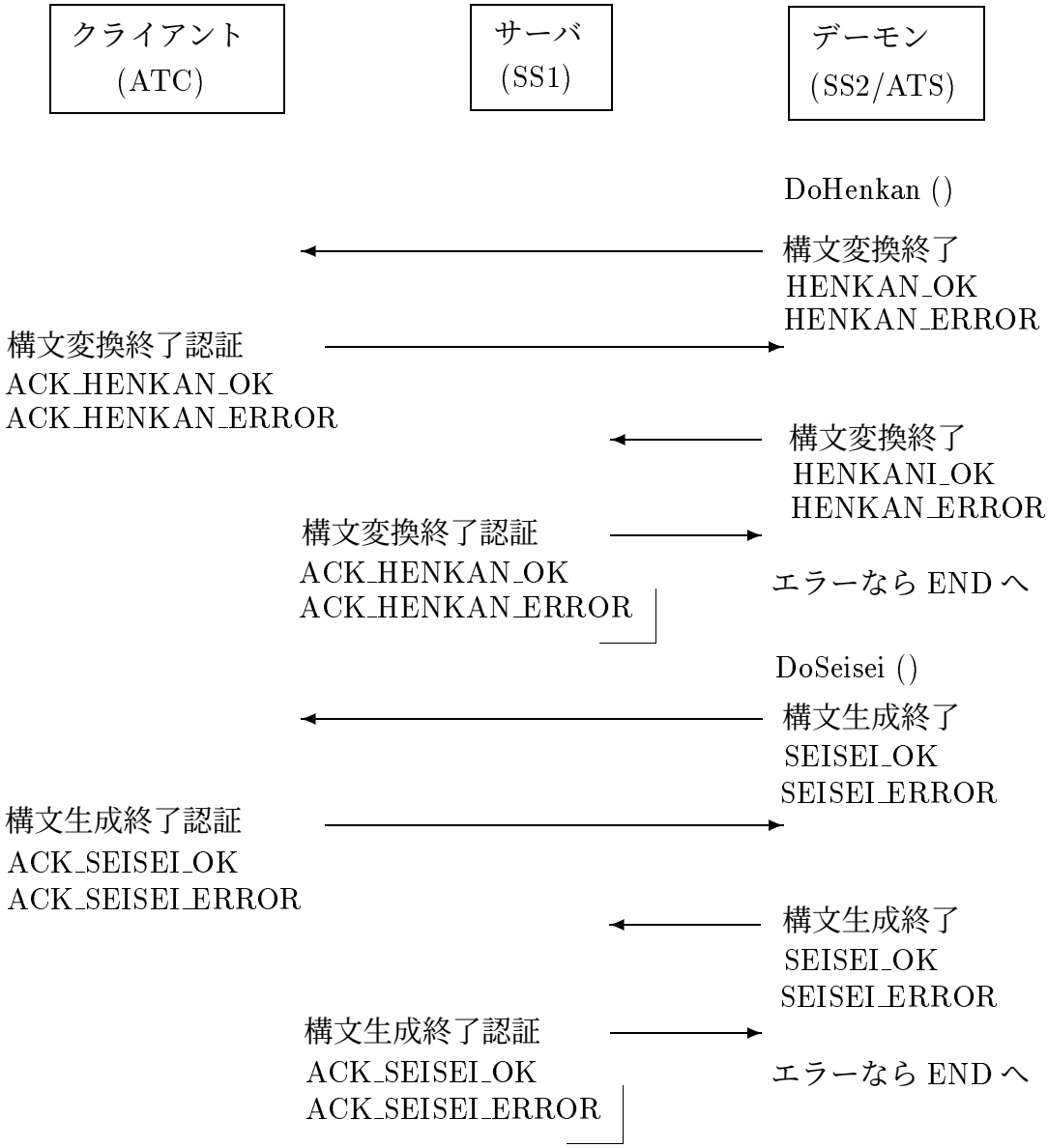

図 5.4: ステートダイアグラム 4

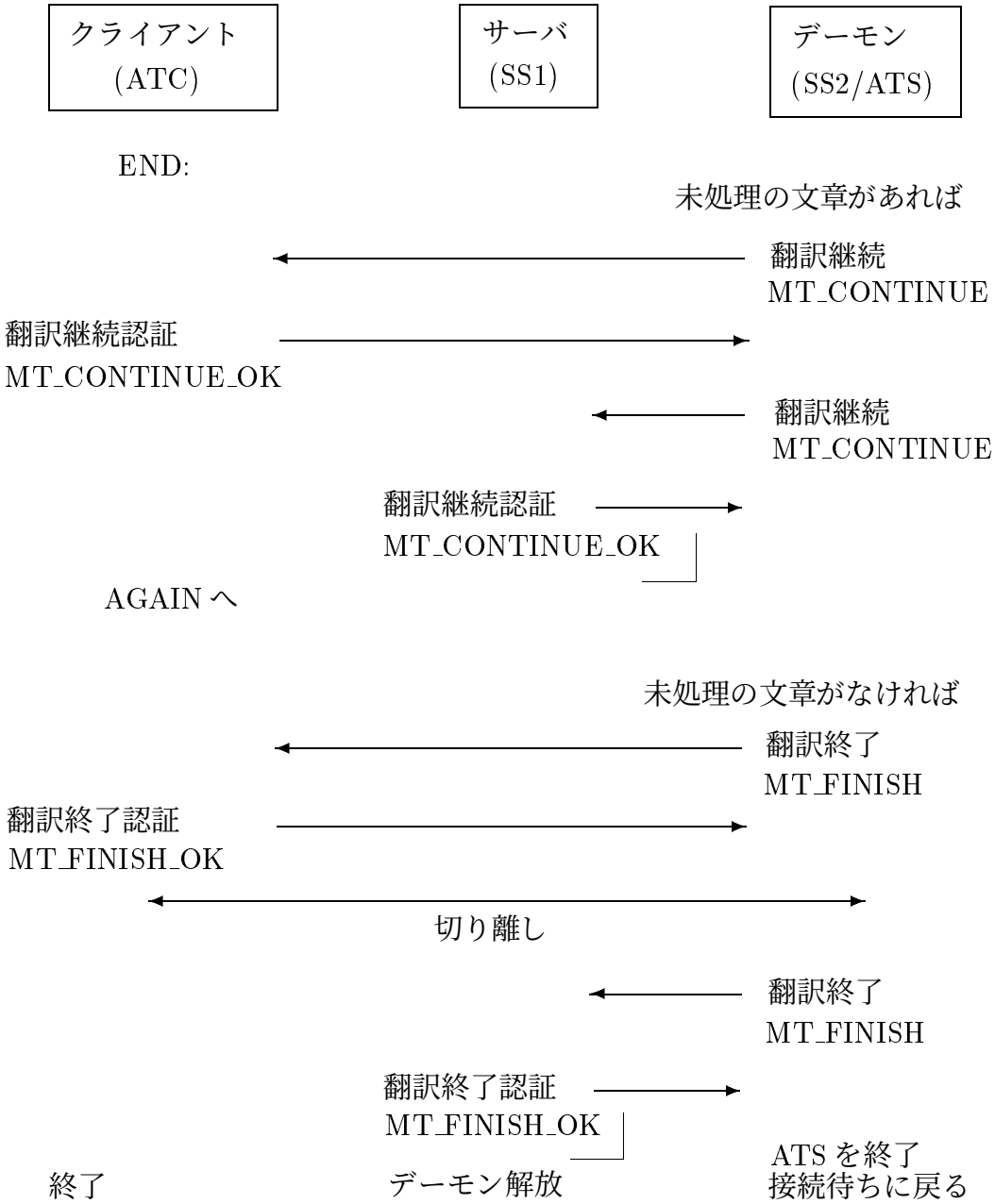

図 5.5: ステートダイアグラム 5

たかも一対一で SS1 と通信を行っているかのように見える。後ろのステートダイアグラムの図もこのように見えるかもしれないが、実際の SS1 は複数の ATC や SS2 に対してタイムシェアリングに一連の処理を行っている。ATC からの要求に対しては、空きデーモンを探してそのホスト名を伝えるまでを１つの処理単位とし、SS2（ATS）に対しては、翻訳処理の４つの行程の途中経過に関する通信や翻訳終了、または翻訳継続を認識するための通信を１つの処理単位としている。４つの行程のうちの１つに関する通信が１つの処理単位である。（図の」で区切られた SS1 の処理が１つの処理単位である。）

SS1 が立ち上がると、次に SS2 がこれに接続を要求する。SS1 と SS2 の間でコネクションが確立されて始めて ATC は翻訳処理を実行することができる。SS1 はシステム全体で受け付ける SS2 の数を限定している。現在は、負荷を考慮して、最大２個までとしている。また、SS1 は接続を許可する SS2 のホストを決めているので、SS2 はどのようなホストでも起動することはできるが、必ずしも SS1 と接続できるとは限らない。ここでの接続とは、単なるコネクションの接続ではなく、通信を確立するための接続を意味する。受け付け可能な SS2 を判断するために、SS1 は SS2 が起動すると、そこからデーモン開始要求のコードと起動しているホスト名を受信する。あらかじめ用意されているテーブル（構造体の配列）にそのホストが存在したらデーモン開始要求認証を SS2 に送信し、そこから本当の意味での接続が確立される。さらに SS1 は、負荷を考えて、１つのホストからは最大１つの SS2 しか接続要求を受け入れないので、すでに接続を確立している SS2 が存在するホストから新たに SS2 の接続要求を受信したときは、接続を許可しない。

ATC は翻訳を開始するとき、まず SS1 に接続要求を送信する。コネクションが確立されたら、次にホスト名、ユーザ名、パスワードを SS1 に送信する。SS1 は送られてきた３つのデータからその ATC が受け付け可能なユーザであるかどうかを判断し、正当なユーザであると判断できるときのみ、その ATC に開始データ認証を送信する。（現時点では、ユーザが入力したパスワードからだけその判断を行っているが、この翻訳システムを構築するネットワークが広域になればなるほどホスト名とユーザ名からもその判断を行わなければならない。）入力されたパスワードが誤っているとき、SS2 はその ATC に開始データ認証エラーを送信し、接続要求を拒否する。SS1 から翻訳システムの使用許可をもらった ATC は、次に SS1 に空きデーモンホストを探すことを要求する。SS1 は、接続している SS2 を順に検索していき、空いているデーモンホストが見つかったらそのホスト名を ATC に知らせる。もしすべてのデーモンホストが使用中であれば、SS1

はそのことを ATC に伝える。ATC は、空きデーモンホストを受信するにしろ、すべてのデーモンホストが使用中であることを受信するにしろ、それに対する ACK を SS1 に送信し、ここで両者の間のコネクションは切れる。今のところ、SS1 は ATC に対する待ち列を用意していないので、ATC は空きデーモンを獲得できなかったら、そこで終了する。

空きデーモンホストを獲得した ATC は、直接 SS2 に接続要求を送信する。コネクションが確立されたら、次にホスト名、ユーザ名、翻訳ファイル名を SS2 に送信する。SS2 はこの３つを受信したら、翻訳データ認証を ATC に送信する。現時点では、SS2 はこの３つを受信しさえすれば ATC に翻訳データ認証を送信しているが、このままだと翻訳ファイル名を送信する意味が全くない。そこで将来的には、この SS2 でも ATC の接続許可を判断し、認めることができないユーザであると思われるときは、ここで接続要求を拒否するようにする。そのために SS2 は ATC にこの３つを送信させているのである。接続要求を許可された ATC は、次に翻訳を実行してもらいたいソーステキストを SS2 に送信する。１つの ATC が長時間 ATS を使用し続けることを避けるため、ATC が１回の ATS の使用で送信できる翻訳ファイルは１つで、その文字数は全部で１０２４文字に限っている。コンソール入力の場合も同様で、ユーザが [RET].[RET] で入力を終了するまでのデータをテンポラリファイルにためておいて、そのファイル１つに限っている。コンソール入力も文字数の上限は１０２４文字である。SS2 はソーステキストを受信したら、ATC にソーステキスト認証を送信する。

SS2 はソーステキストを受信したら、Lisp を立ち上げて ATS を起動する。このとき、SS2 と ATS は双方向パイプで結ばれる。それから、SS2 はソーステキストを１文ずつ区切って翻訳を実行していく。始めに形態素解析が行われる。このとき SS2 が呼び出す関数は、引数としてソーステキストの１文をとり、それとともにこの関数が呼び出されると、ATS は双方向パイプを通って送られてくるその文章に対して形態素解析を実行する。次に SS2 がそれの結果を得るための関数を呼び出すと、ATS は双方向パイプを通して SS2 にその結果を送信する。SS2 はその結果を受信したら、直接 ATC に形態素解析終了を表すコードとともにその結果を送信する。このコードはその結果が正常なものであるかエラーを表すものであるかを示している。ATC はそれを受信したら、そのコードに合った ACK を SS2 に送信する。SS2 はそれを受信したら、今度は途中経過を知らせるために、SS1 に形態素解析終了を表すコードだけを送信する。SS1 はそれを受信したら、そのコードに合った ACK を SS2 に送信する。ここで結果がエラーであったら、ATC も SS2（ATS）もこの文章に対する翻訳処理を終了する。ATS

は Lisp 上で稼働するシステムだから、４つの行程の結果もそれ自身が理解できるものとユーザが理解できるものとはかなり違いがある。そこで ATS は SS2 が翻訳処理結果を要求してくると、その結果をユーザが理解できるような文字列になおして送信する。このとき、ATS は自分が理解できる結果は保持しておいて、次の行程を実行してほしいということを示す関数が SS2 で呼ばれるまで待機している。従って、構文解析、構文変換、および構文生成を ATS に実行させるための関数は、引数を取らないのである。同様にして、エラーが起こらない限り順に構文解析、構文変換、および構文生成の一連の流れを実行していく。ATS で Lisp でのエラーが生じたら、特別にそのことを示すメッセージとコードを送信する。

翻訳の結果が正常に得られても、エラーが得られても、１文の処理が終了したら、SS2 は未処理の文章が残っていれば翻訳継続を表すコードを ATC に送信し、未処理の文章が残っていなければ翻訳終了を表すコードを ATC に送信する。ATC はそれを受信したら、そのコードに合った ACK を SS2 に送信する。SS2 が翻訳終了の ACK を受信したら、この時点で ATC と SS2 の間のコネクションは切れる。同様に SS2 は SS1 に対してもこれを行い、SS1 も同様に ACK を送信する。SS2 が翻訳継続を ATC および SS1 に送信したら、全体の動きはまた形態素解析から実行される。また、SS2 が翻訳終了を ATC および SS1 に送信したら、ATC は動作を終了し、SS1 はデーモンを空き状態にして別の要求の実行に移り、SS2 は ATS を終了させて再び接続待ちの状態に戻る。

5.2　DCP 方式の実装における特徴と問題点

DCP 方式による広域分散型機械翻訳システムには、いろいろな特徴がある。利点であることも多いが、逆に欠点であることも少なくない。このセクションでは、これらを具体的に考えて、現時点ではどこまで実現できるのかを明確にし、問題点は何なのかをはっきりさせることにする。

5.2.1　スケジューリングサーバ（SS）の並列処理

広域分散型機械翻訳システムは ATC と SS と ATS の３つのプロセスから成り立っているが、その中で通信という点から一番重要な役割を果たしているのは SS である。前のセクションでも述べたように、DCP 方式では SS1 と SS2 という２つの SS が存在するが、ここで述べる SS とはシステムの中核をなす SS1 の方である。DCP 方式の SS1 は、複数の SS2 と ATC を管理しなければならない。SS1 と SS2、SS1 と ATC、および SS2 と ATC の間は、すべてコネクションを張って通信を行う。つまり TCP を

使用する。UDP を使用すればあまり問題なく複数の相手からの通信を扱うことができる。これはコネクションを張っていないので、とにかく到着した要求を順番に処理して、アドレスを使ってその相手に返せばよい。しかし、正確に処理を行うことを何よりの目標としている今回の実装においては、やはり信頼性のない UDP を使うよりも、信頼性のある TCP を使用することを選択する。TCP による通信において、複数の相手を扱う方法はいくつか考えられるが、ここでの実装においては、select() という関数を使うことによって、複数のソケットをタイムアウトを決めて順番に処理する方法を採用した。そこで、少しこの方法を説明することにする。

SS1 はまず、起動して待機状態に入っているときに必要なソケットと、新しい ATC あるいは SS2 からの接続要求を受け付けるためのソケットと、SS2 から接続要求を受信したとき複数の SS2 を識別するためのソケットの配列を用意する。その配列の大きさは、SS1 が接続を許可する SS2 の数＋１個である。最後の１個は、SS1 が空いている SS2 を探すときに使用するもので、それほど重要なものではない。そして起動して待機状態に入ったら、その直後から、待機状態に入っているソケットと SS2 を識別する複数のソケットをタイムアウトごとに順番にセレクトして、要求が到着していたらその処理を行う。このときに行う処理の大きさは、他の ATC や SS2 に迷惑がかからない程度の量を１つの処理単位とする。

複数の通信相手を処理する方法として他に考えられるのは、SS1 をフォークするやり方である。しかしこのやり方だと多少不都合なことが生じる。まず、子プロセスの動きをどのように親プロセスに知らせるかである。SS1 は、ATC と SS2 の両方からの接続要求を処理する。一般にサーバがフォークする時期は、クライアントからの接続要求を受信するときである。ここで、SS1 がある SS2 から接続要求を受信したときにフォークすると仮定する。すると、子プロセスはその SS2 とコネクションを確立し、親プロセスはソケットを閉じて別の接続要求を待機することになる。ここでもし ATC から接続要求が到着したら、親プロセスは今接続を要求してきた SS2 が動いているものと判断してこの ATC をそこに割り当てる。このときは、ATC に SS2 が割り当てられたらコネクションが切れるので、わざわざフォークする必要はない。ところが、この SS2 が SS1 の認めていないホストであったとすると、SS2 が SS1 の子プロセスにデーモン開始要求を送信したら、その時点で接続を拒否されて、両者の間のコネクションは切り離されてしまう。これを親プロセスが認識できないところが問題である。なぜならば、フォークした時点から親プロセスと子プロセスは全く別のプロセスになってしまうからである。従って、当然 ATC はこの SS2 に接続できない。これを解決するために、フォークしたら子プロセスと親プロセスの間に双方向

パイプをつなぐことも考えられる。しかし、このようにすれば、結局親プロセスはこの双方向パイプの窓口と新たな接続要求の窓口をセレクトしなければならないので、実装しているやり方と変わらない。もう一つフォークした場合の問題点は、メモリをたくさん消費することである。込み入ったマシン上で SS1 を動かさざるおえない状況になったときに、この方法だと困ってしまう。

フォークせずにセレクトを使う方法も問題がないわけではない。タイムシェアリングに複数の要求を処理する以上は、やはりどのような処理をどのように扱うかをきちんと決める必要がある。この DCP 方式による広域分散型機械翻訳システムは、パスワードを入力させて使用許可の判断を行っている。ここでもしパスワード入力にタイムアウトを用いないと、いじわるなユーザが永遠にパスワードを入力しないと SS1 は ATC から送られてくるパスワードを読むところでブロックしてしまう。これは、システム全体のブロックを意味する。そこで今度はパスワード入力に１０秒のタイムアウトを設定する。これならば、１０秒以上の SS1 のブロックはありえない。ところが、ここで ATC がパスワードを入力するまで SS1 が他の要求処理に移らないとすると、いつもパスワードの送受信のところでシステム全体がブロックするので、かなりの遅延が生じてしまう。これを避けるために、SS1 は ATC からホスト名とユーザ名を受信したら、一度別の要求処理に移り、ある程度経ってからパスワードの受信に入るようにするべきである。このように、SS1 の１つの処理の切れ目と扱い方をしっかりさせることは、翻訳システム全体をしっかりさせることにつながるのである。

5.2.2　自動翻訳のためのクライアント（ATC）のラップタイム表示

ATC も SS も ATS も起動するときに'T' というオプションを指定すると、通信の一連の流れをウィンドウに出力する。指定しなければ SS と ATS は何も表示せず、また ATC は翻訳処理結果と以下で述べるような翻訳処理時間だけを表示する。

ATC は T オプションを指定したときに、SS1 からの接続要求認証を受信した時点を０として、そこからの実行時間をはかり、何かを受信するごとにラップタイムを表示するようにしてある。４つの行程のところでは、ラップタイムだけではなくて、その行程にかかった処理時間も表示する。この処理時間は、オプションに関係なく表示する。これによってどんな文章がどれくらいの時間で翻訳されるかがわかる。この時間を短縮するためには、やはり翻訳システムの本体（ここでは ATS）を高速のスーパコンピュータで稼働させることがまず思いつく。SUN workstation の上で動くこのシステムをスーパコンピュータ上で動かすためには、翻訳システムそのものを

いじる必要があるので、ここではこのような案があるということにとどめておくことにする。

5.2.3 バッファサイズにおける問題点

どのようなクライアント／サーバモデルでも必ず考えなければならないことは、バッファサイズである。現段階では、ATC も SS も ATS もバッファを用意するところはすべて２０４８バイトに設定してある。まだ十分なテストを行っていないため、どのような文章がこの大きさでオーバフローするかははっきりしていないが、いずれにしろどんな大きさのデータでも送受信を行えるようにしなければならない。この３つのプロセスのあらゆるところでデータの送受信は行われているが、やはりその都度必要なメモリ領域を送信側と受信側に確保させて通信を行うことが、最も自然なやり方である。送信側は受信側にまず送るデータサイズを送信する。受信側はそれを受け取ったら、それに合うだけのメモリを malloc() によって確保する。このようにすれば、空きメモリがある限り、大きなデータの送受信も可能である。また、バッファサイズごとに分割して送受信を行うことも考えられる。第３章で述べたクライアント／サーバモデルのサンプルプログラムにおけるバーチャルサーキット型はこの方法を使って送受信を行っていることはすでに述べた。いずれにしても、このような機能は絶対に必要なことである。

5.2.4 ACK の通信方法

通信の流れを示した前のセクションからもわかるように、DCP 方式では、あるデータの受信に対して必ず ACK を返すようにしている。これは ATC、SS、ATS のすべてにおいてそのようにしている。この ACK の送受信にもいろいろな方法がある。ここでは、これについて述べることにする。

ACK の送受信方法は、コネクションのはり方にも依存してくる。実装した DCP 方式は、ATC と SS1 の間は ATC が SS1 に空きデーモンホストを要求し、そのホスト名を受信するときだけコネクションが張られる。実際に翻訳処理が行われているときは、ATC と SS1 はコネクションで結ばれてはいない。そのかわりに、SS1 が起動してから、それが SS2 からのコネクト要求を受け付けて、SS1 と SS2 の間でコネクションを確立しているのである。このコネクションを確立する方法は、これとは逆に SS1 から SS2 の方へコネクト要求を出す方法が考えられるが、この方法だと、SS2 がダウンしたら再起動するときにまた SS1 を起動するところから始めなければならない。つまり、SS1 を一度止めなければならない。これだとかなり不便である。よって、SS1 が SS2 を受け付ける形式を採用した。これならば、

SS2 がダウンしても、また SS1 にコネクト要求を出せば、容易に再起動できる。また、起動する SS2 の数も調節することができる。もっとも SS1 がダウンしたら、どのような方法を採っていても同じである。

ATC、SS1、SS2（ATS）のうちの二者間での ACK の送受信方法は一通りしかない。ここで考えるのは、三者間での ACK の送受信方法である。問題になるところは、翻訳処理の結果を SS2 から ATC に返送するときである。実装した DCP 方式は、SS2 が ATC に結果とそれを表すコードを送信すると、それを受信した ATC は SS2 にその ACK を返送し、SS2 はそれを受信すると、次は SS1 に結果を表すコードだけを送信し、それを受信した SS1 は SS2 にその ACK を返送する。つまり、SS2 が SS1 と ATC の通信相手をしている。従って、ATC と SS1 の間は不必要にコネクションを張っておくことはしない。これとは違って、一連の翻訳処理が終わるまで ATC と SS1 の間のコネクションを張り続けておいて、ATC が SS1 と SS2 の通信相手をする方法もある。このときは、SS1 と SS2 の間にコネクションを張る必要がない。しかし、この方法では、ATC にかなり負担をかけることになる。また、SS1 が SS2 の状態を知ることができるかどうかも ATC にかかってくる。このように、クライアントに必要以上の負担をかけるようなクライアント／サーバモデルは、あまり望ましいものではない。従って、通信の責任を担うところは、SS1 か SS2 になるようにしている。

5.2.5　自動翻訳サーバ（ATS）のためのメモリ確保

前にも述べたように、ATS は立ち上がるために莫大なメモリを必要とする。これを常に確保しておくかどうかが少し問題である。もし、ATS のために必要なメモリをいつも確保しておくとすると、ATS が立ち上がっていないときには、このメモリはすべて無駄になっている。ATS を一度立ち上げたら永久にそのままにしておくようにしても、ユーザが使用していないときには同じようなことが言える。そこで、メモリを気にせずに立ち上がれたら ATS を立ち上げて使用することにする。つまり、ATC は空いている SS2 を割り当てられても、SS2 は ATS を立ち上げられるかどうかがわからないのである。今のところは、これに関して制御することはしていない。ATC がある SS2 を割り当てられて、この SS2 が ATS を立ち上げられなかったら、当然その時点で翻訳処理が終了する。しかし、次の ATC が接続要求を SS1 に送信したら、またこの SS2 が割り当てられてしまう。だから本来ならば、SS1 が定期的に ATS を立ち上げられるだけのメモリ領域を確保できていない SS2 を調べて、このホストを使用停止状態にし、使えるようになったら、また使用可能状態にすればよい。これも将来的には必要なことである。

5.2.6 入力文字について

日英翻訳で入力する日本語は全角文字である。半角文字を入力すると、システムは変換を行うことができない。また、コードは EUC コードである。その他のコードは変換することができない。SS2 は、ATS に ATC からのソーステキストを送る前に、それが正当なものであるかを判断しなければならない。EUC コードであるかどうかを調べるのは大変なので、ここでは半角文字が交じっていないかどうかだけを調べるようにする。EUC コードにすることは、ユーザに任せるようにしているが、これも将来的には制御しなければならない。SS2 は、ATC から送られてくるソースデータのうち改行以外の半角が入っているかどうかを調べて、入っていたら、ATC にエラーコードを返送するようにしている。ATC はこれを受信したら、その時点で異常終了する。

第 6 章

評価と結論

第４章と第５章で、広域分散型機械翻訳システムをどのように設計しネットワーク上に実装するかについて述べてきた。この章では、実際にそのシステムを使用するユーザから見て、これがどのように感じるか、またどこをどのように変えていかなければならないかについて述べることにする。

6.1 広域分散型機械翻訳システムの利用効果

実際にいくつかの日本語の文章を翻訳してみた。入力した文章とその翻訳結果は以下のとうりである。使用した辞書が証券関係のものであるから、その方面の文章が中心になっている。

（例１）CSK が値を上げた。

（結果）CSK Co. rose.

（例２）これは本である。

（結果）This is a book.

（例３）今日は円安ドル高の一日だ。

（結果）Today is the 1st of a yen's weakness strong dollar.

（例４）為替、株、債券のトリプル安の波乱場面では腰の入った買い物は期待しにくい。

（結果）Buy orders that trading was activated hardly expect [due to] wild fluctuations stage of the exchange markets, the stock and bonds of falls.

現状の機械翻訳では、同じような文章を入力しても、機械が判断できる文章は限られているので、必ずしもすべての文章が翻訳できるとは限ら

ない。例えば、上の例２のように「これは本である。」と入力すれば正しい結果を得られるが、「これは本です。」と入力しても正しい結果を得られない。「です」という言葉を別の意味に解釈するからである。従って、実際に機械翻訳を仕事として使用する人たちは、入力する文章が機械の理解できるものであるかどうかをチェックして、手直しを加えてからその文章を機械翻訳にかける。

章末に実際の機械翻訳の流れがどのようになるかを示したウィンドウの出力図を載せた。Figure6.1、Figure6.2、Figure6.3は、例４の文章が翻訳されるときの一連の流れである。４つのウィンドウのうち、右上が SS1、右下が SS2（ATS）、左下が ATC である。Figure6.4、Figure6.5は、例２の文章が翻訳されるときの一連の流れである。Figure6.6は、ATS がダウンしたときの状態である。Figure6.7は、ATC のパスワード入力がタイムアウトしたときの状態である。Figure6.8は、翻訳処理結果がエラーで終了したときの状態である。Figure6.9は、ATC が半角文字を入力しようとしたときの状態である。

次に、上の４つの例文を翻訳するときにどれくらいの時間がかかったかを示す（Table6.1参照）。表の値は５回の平均値で、単位は秒である。また、SS1 のセレクトタイムアウトは１秒である。DCP 方式による翻訳システムは、文章が入力されたときに ATS が立ち上げられ、翻訳が終わると ATS が終了する。同じことを繰り返しても、そのたびに ATS を立ち上げなければならないので、５回とも処理時間はだいたい一定している。また例１と例２を見てもわかるように、長さがほぼ同じである文章において、例１は形態素解析の処理時間が例２の半分だが、構文解析の処理時間は例２の２倍かかっている。従って、処理時間というものはもちろん文章の長さも関係してくるが、それ以上に文章の種類が関係してくる。

表 6.1: 機械翻訳の処理時間

	Keitai	**Kaiseki**	**Henkan**	**Seisei**	**SUM**
Ex.1	4.4	5.8	3.8	2.4	16.4
Ex.2	8.8	2.6	8.4	1.6	21.4
Ex.3	4.4	8.8	7.2	2.6	23.0
Ex.4	24.8	29.2	34.6	7.6	96.2

辞書の確立さえしっかりさせれば、口語の文章の機械翻訳もかなり実用化される。また、ここでのクライアント／サーバモデルの形式は、他のものにも応用できるので、幅広く利用していきたい。

6.2　今後の課題

第４章で述べた翻訳環境モデルのうち本研究で実装したものは、DCP 方式によるものである。この実装において何よりもまず達成しなければならなかったことは、１つの大きな翻訳システム環境を複数のユーザで効率よく利用できるようにすることであった。そのためにスケジューリングサーバを用意した。また、より正確な通信の流れを確立することも重要であった。この２つは、ある程度のレベルでは達成されたように思われるので、これからまず考えなければならないことは、この DCP 方式のシステムを DCC 方式のシステムへと拡張することである。最初にバッファのオーバフロー制御を行わなければならない。これの解決策については第５章で述べた。それから、翻訳システムの４つの行程が独立に動くようにするために、Lisp で記述された翻訳システムそのものを改良しなければならない。この４つの行程がきちんと４つのプロセスとして働くようになると、SS1 と SS2 の両方のスケジューリングサーバをもう少ししっかりとさせる必要がある。１つのホストで動く ATS に複数のユーザを効率良く割り当てられるようにするために、待ち列を設定することが大事である。システム全体の遅延をできる限り少なくするためには、これはかなりしっかりとさせなければならない。

ATS が翻訳の処理を行ってその結果を SS2 に送信し、それを ATC が受け取ったら、現時点ではそれをそのままディスプレイに出力している。構文解析と構文変換の結果は、文法を記述するデータが画面上に現れる。この２つの結果は、本来ならばきちんとした木構造で画面に出力させなければならないものであるが、これはかなり時間のかかる作業で、しかも本研究の趣旨からは少しはずれているように思われたので、これに関しては手をつけなかった。しかし、このユーザインタフェースの部分は非常に大事であるので、機械翻訳というものをよりわかりやすくするためにはこれが重要である。

最後に、一番真剣に今後の課題として取り組まなければならないものとして、Translation Phone のことをあげたい。本研究では、時間の都合上、日英翻訳しか扱わなかったが、やはりこれを行うためには英日翻訳も必要である。おおまかな原理は両方とも同じである。また、Translation Phone は口語の辞書の確立や翻訳速度の高速化などいろいろな問題を含んでいるが、これからのネットワークにおいて必要不可欠なものであると思う。国際通信の簡易化というものが、やはりネットワークでの何よりの目標であると思うからである。

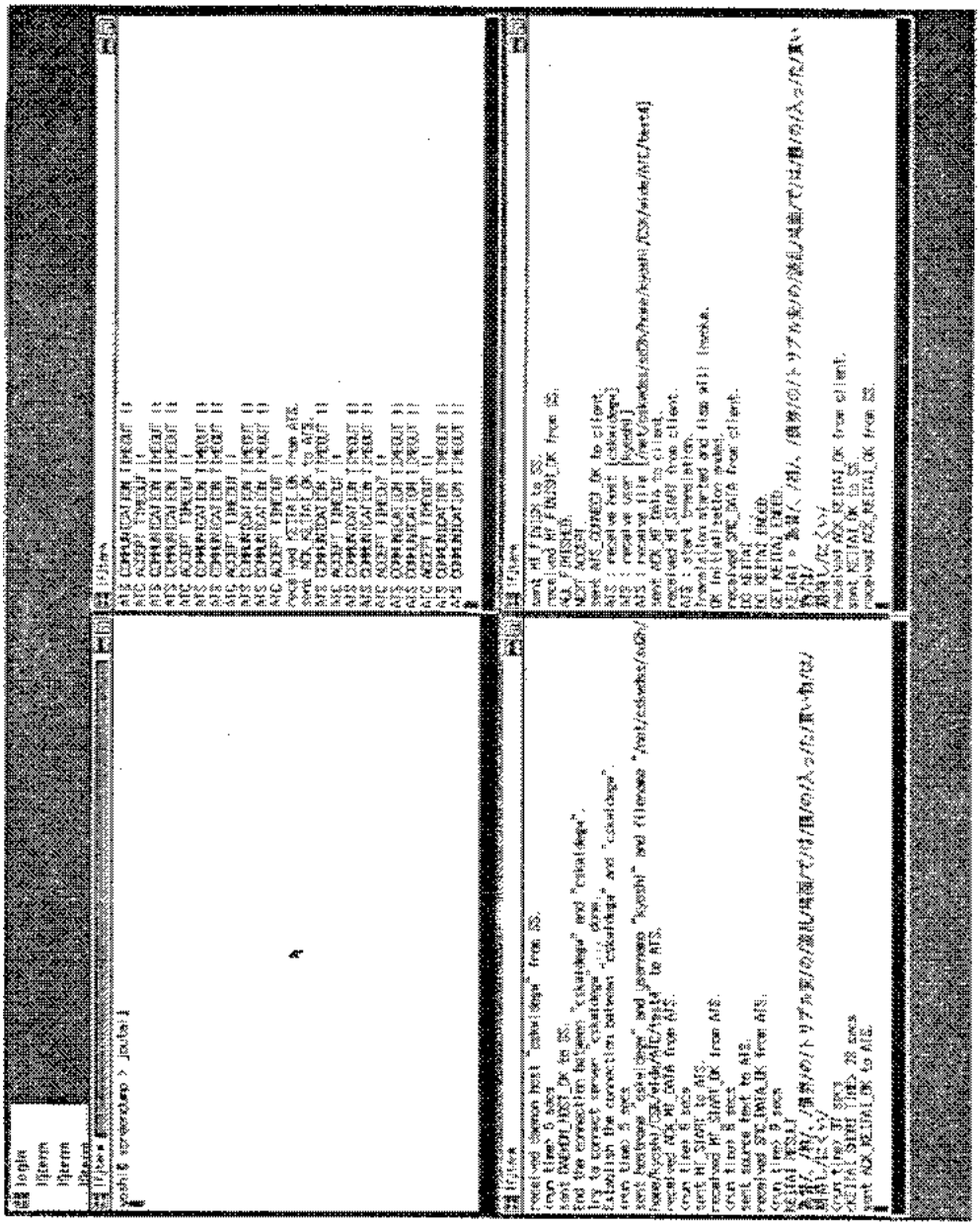

図 6.1: 機械翻訳の実行例 1

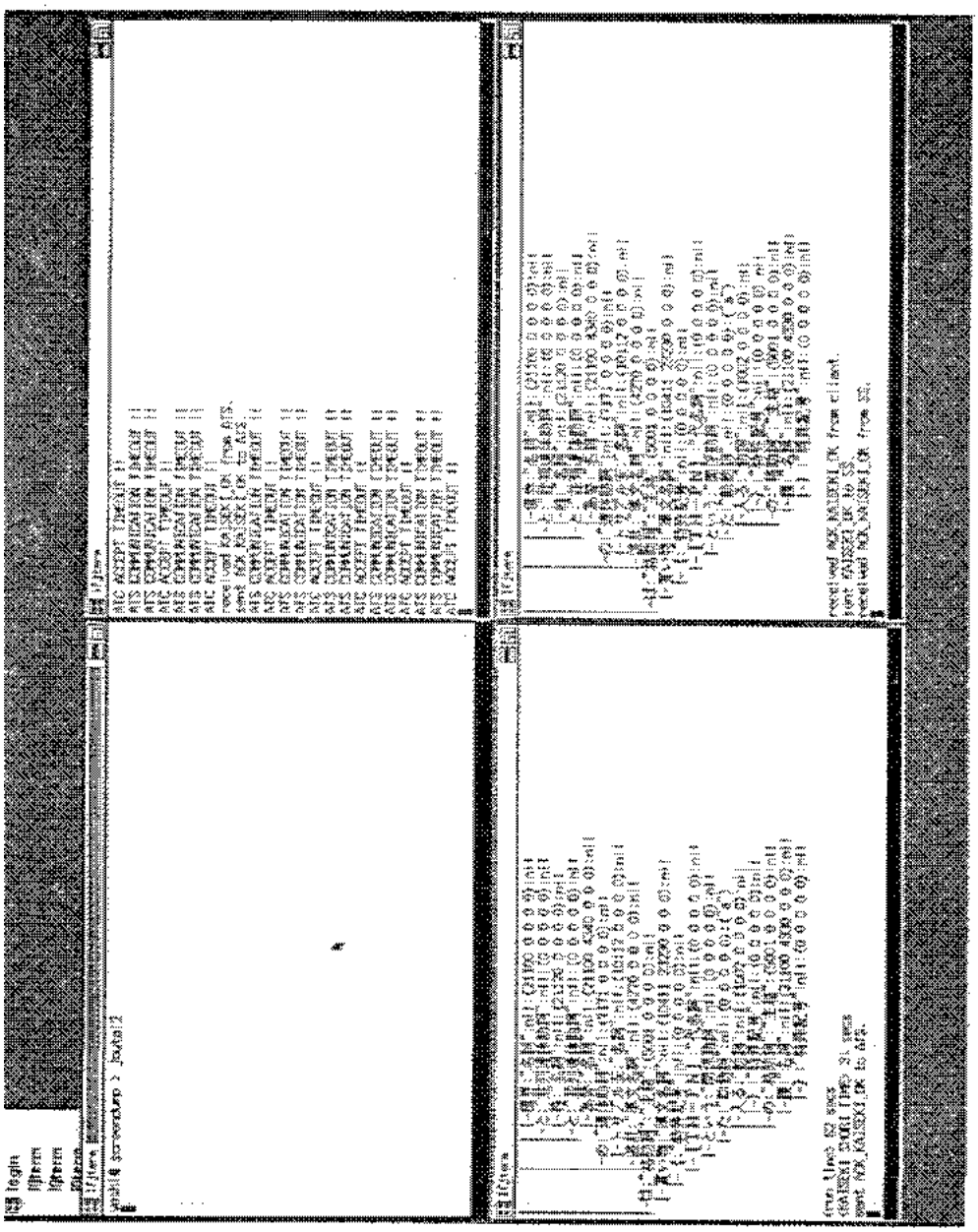

図 6.2: 機械翻訳の実行例 2

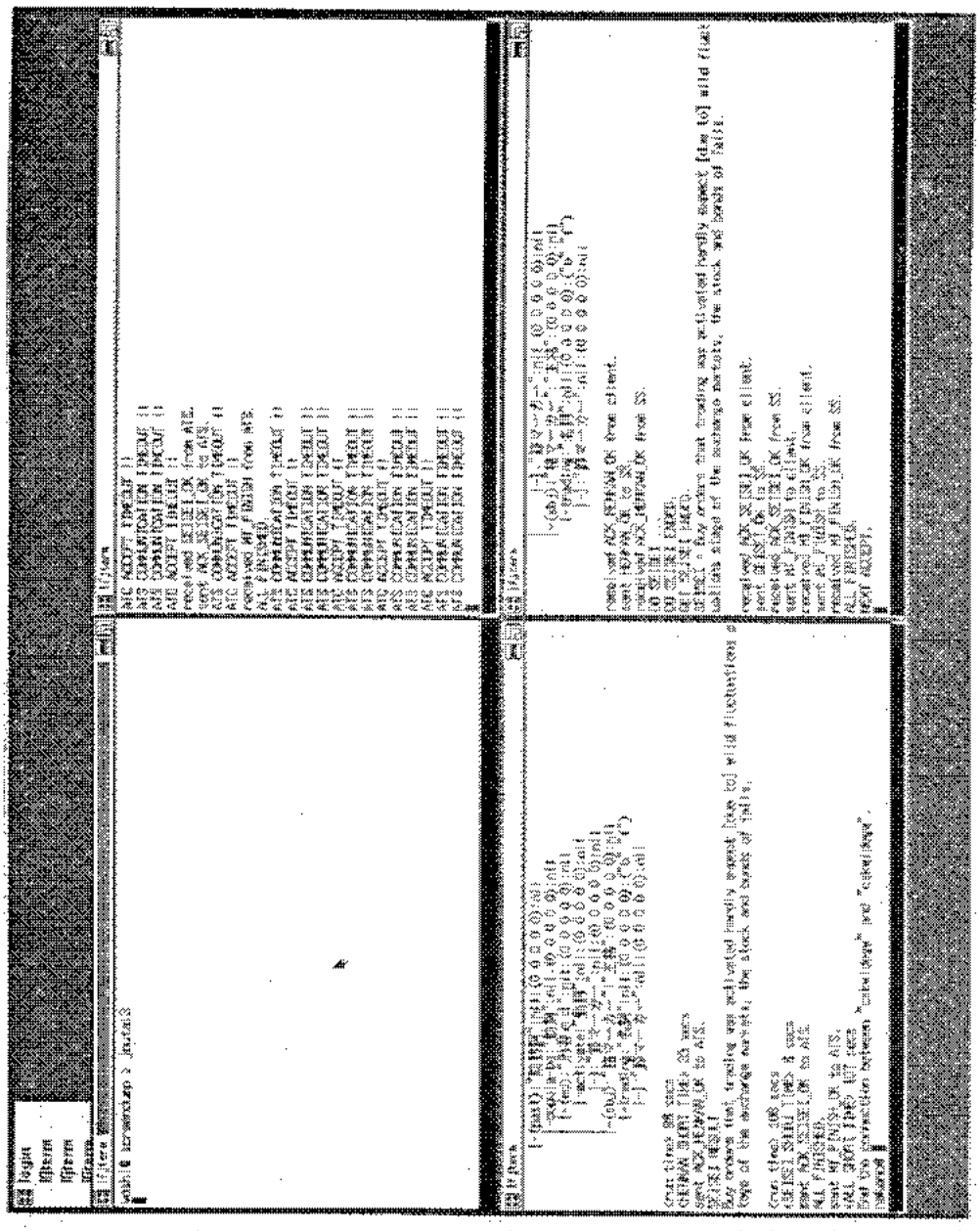

図 6.3: 機械翻訳の実行例 3

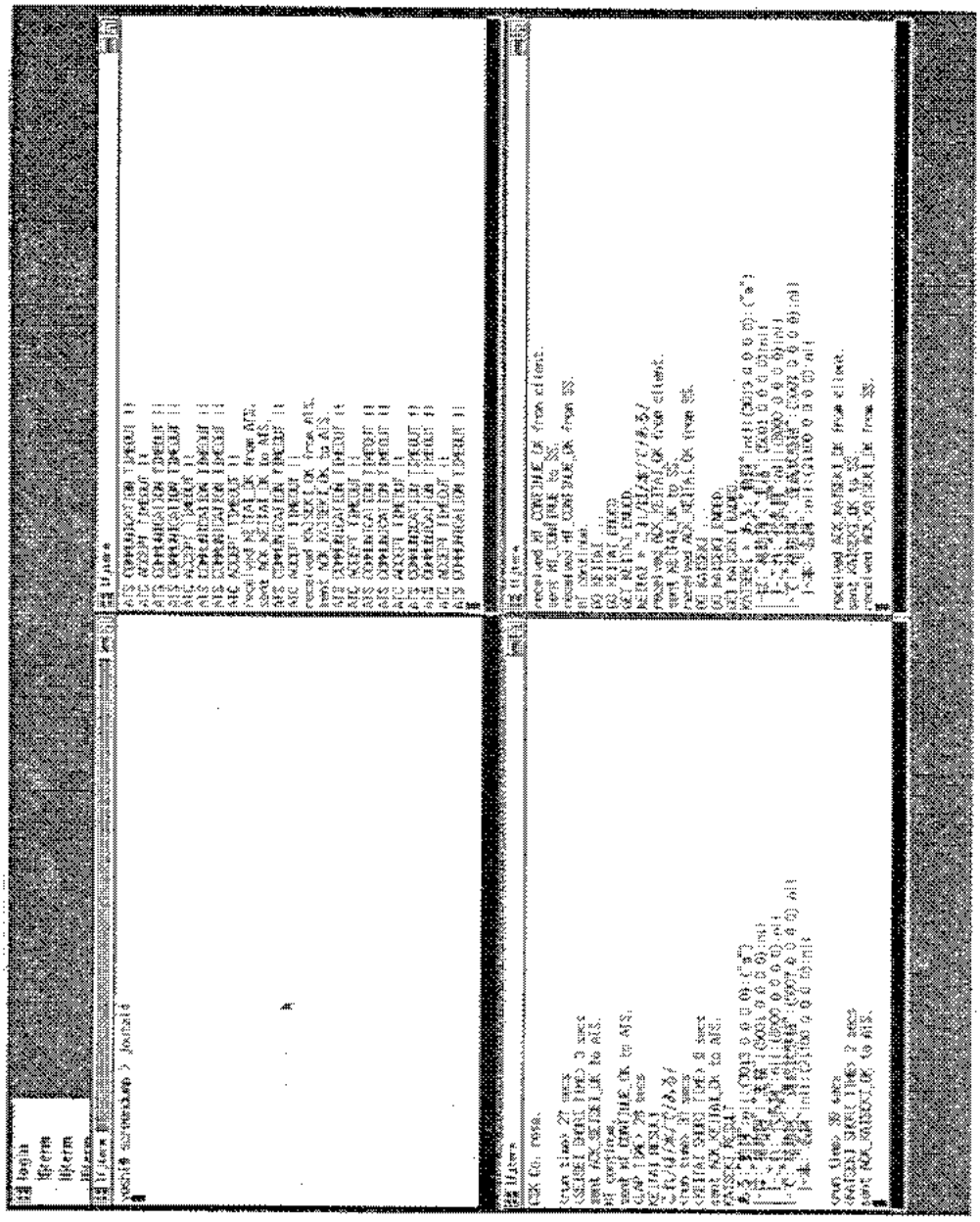

図 6.4: 機械翻訳の実行例４

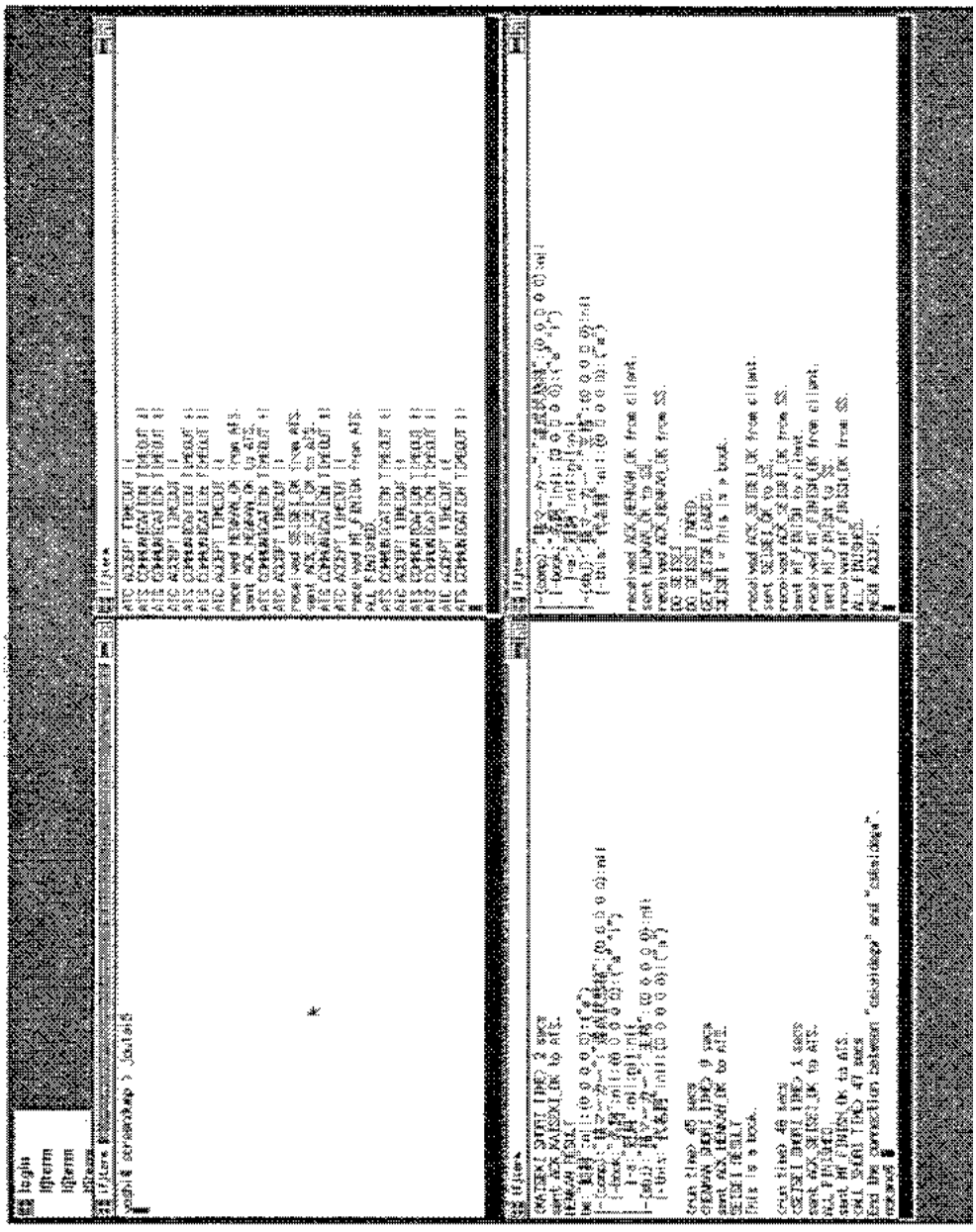

図 6.5: 機械翻訳の実行例５

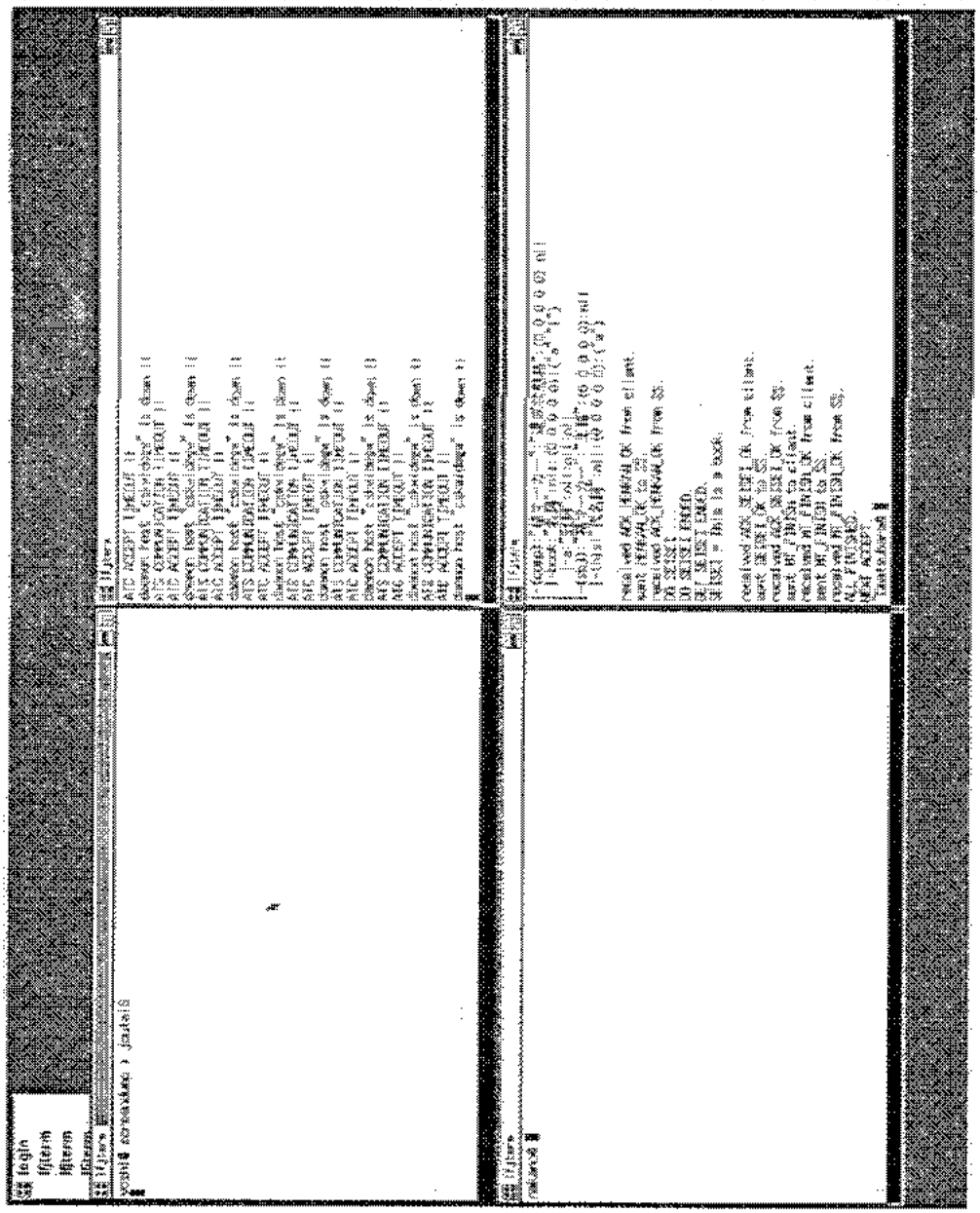

図 6.6: 機械翻訳の実行例６

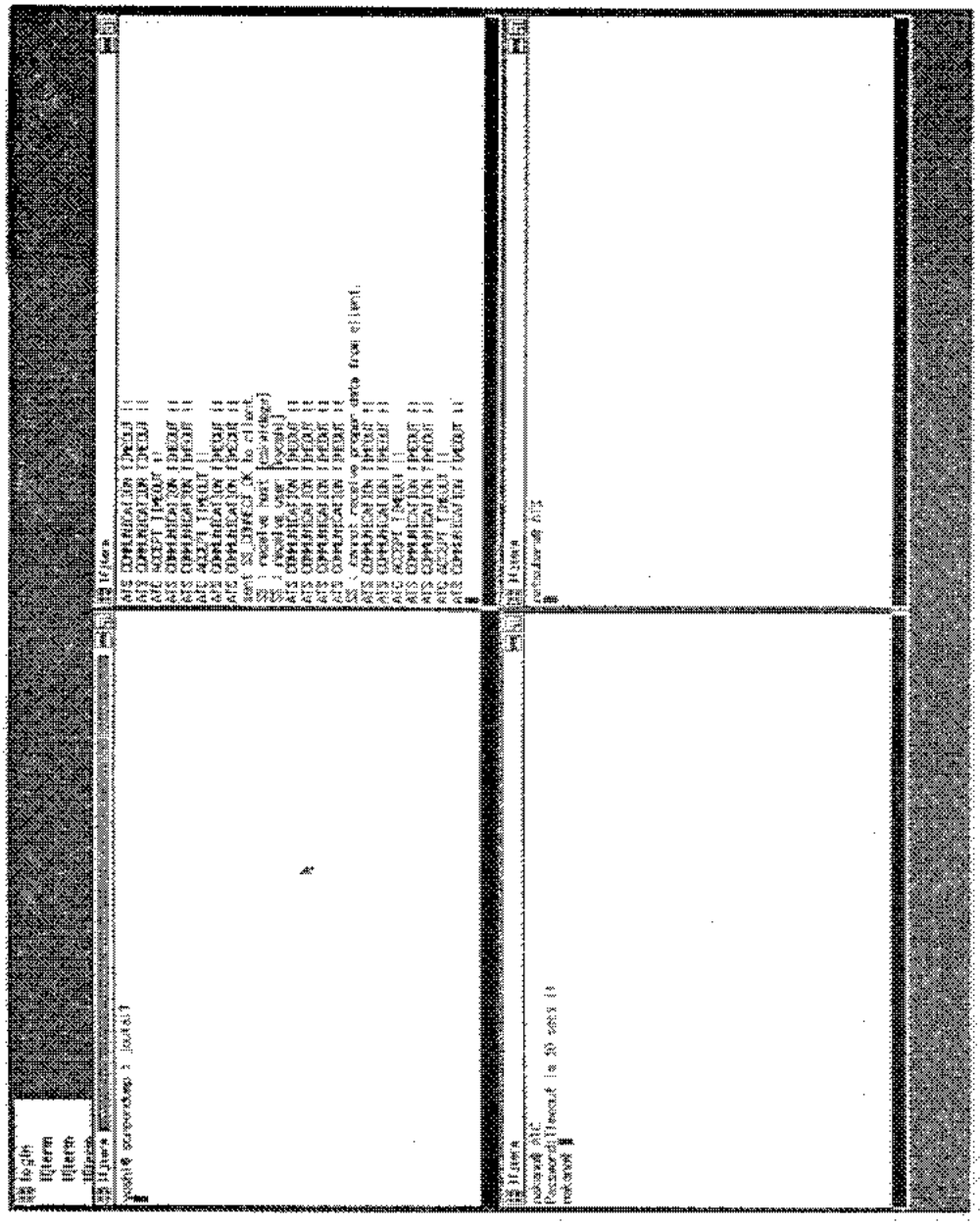

図 6.7: 機械翻訳の実行例 7

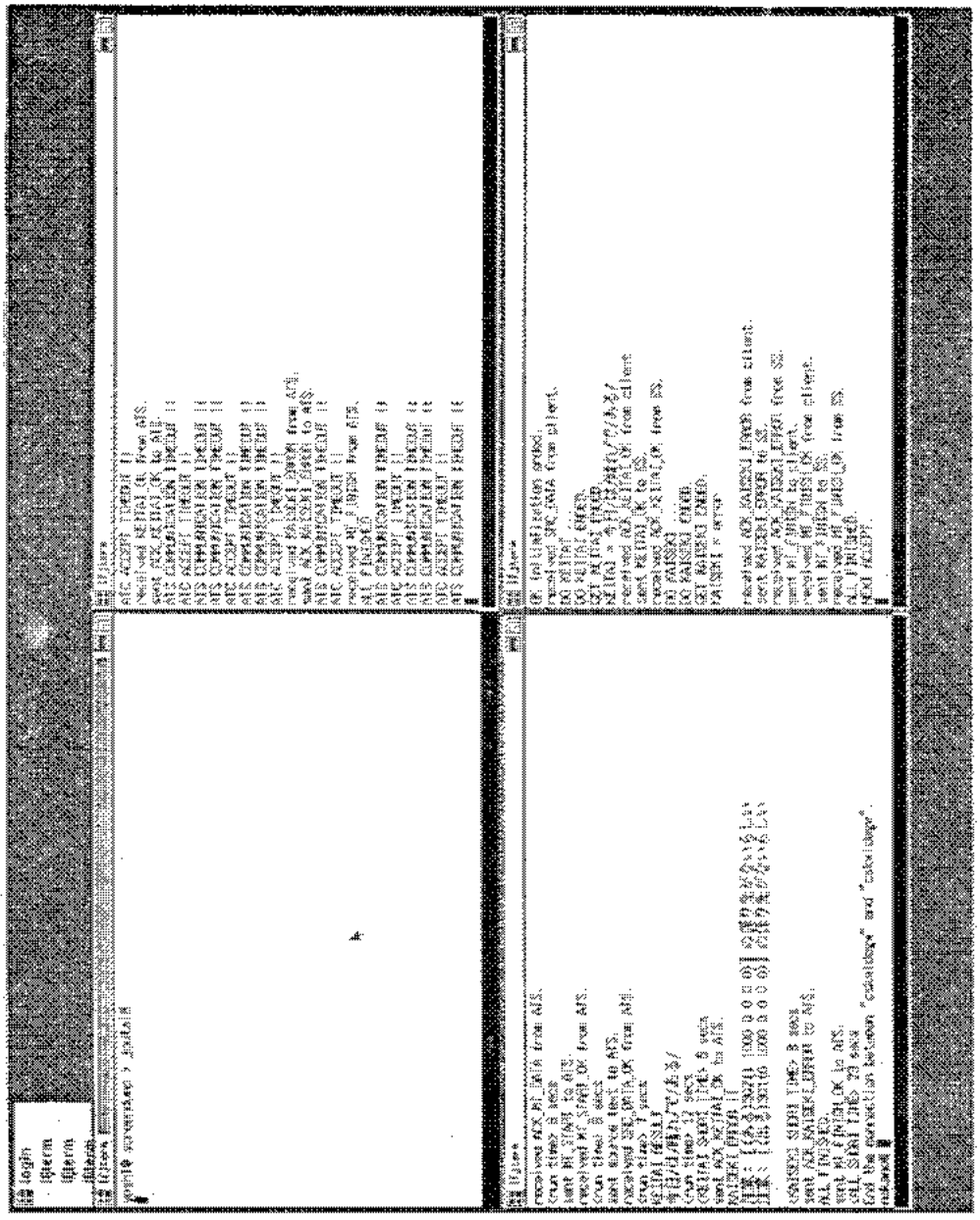

図 6.8: 機械翻訳の実行例８

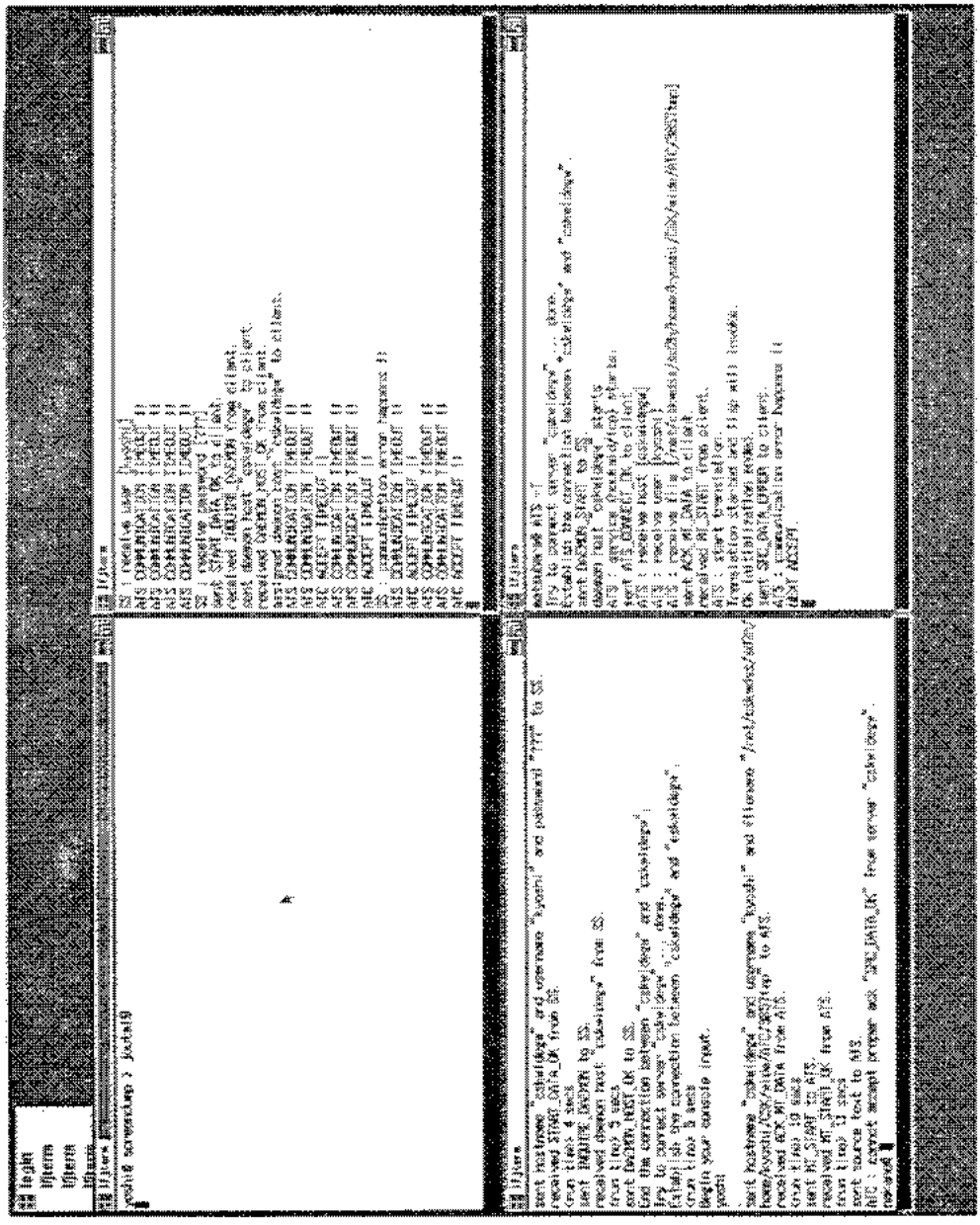

図 6.9: 機械翻訳の実行例９