平成16年度 学士論文

状況に応じて適切な例外処理を行えるアスペクト指向分散環境実験の支援ツール

東京工業大学 理学部 情報科学科
学籍番号 01-0864-8
熊原 奈津子

指導教員
千葉 滋 助教授

平成17年2月7日

概要

分散環境における実験では例外処理が重要である。なぜなら、分散環境では複数のマシンを利用するため、一台のマシンで実験を行う時と比べて、マシンの故障やネットワーク切断などの例外が発生しやすくなるからである。例えば、マシンの台数が揃っている事を前提とした実験では、一台のマシンが故障していた場合、実験に多大な時間を費やしたにも関わらず得られた結果は意味のないものとなってしまう。状況を修復することができるならば代替可能な他のマシンに置き換えるなどして自動的に修復して実験を続けたい。

しかし、状況に応じて適切に例外処理を施すのは困難である。というのは、同じ例外が発生した場合でも、発生した状況によって、例外処理の内容が変わってくるからである。例えば、発生した例外は同じであっても、異なるホストで例外が発生した場合、施したい例外処理も異なってしまうということである。また、同一ホストで異なる例外が発生した場合でも施したい処理の内容は変わる。シェルスクリプトの内部や Java 言語の `try-catch` 文で例外処理を記述する際、起こりうる例外を細かく指定して、その部分に処理内容を逐一記述すると、例外処理記述がプログラムロジックを記述したメインプログラムのあちこちに散在してしまうことになる。その結果、メインプログラムが見づらくなる他、処理内容を変更した場合には変更しづらくなってしまう。それとは逆に、例外処理をまとめて書こうとすると、細かく制御を指定するのが困難になってしまう。シェルスクリプトにおいては終了ステータスは単なる整数であり、例外に関する情報が少ないため効果的な修復が難しい。Java 言語を拡張した汎用的なアスペクト指向言語 AspectJ を用いると例外処理記述をメインプログラムから分離して記述することができ、細かく制御することも可能だが、例外処理はメインプログラムに強く依存してしまう。そのため、メインプログラムの中身を理解しないと例外処理が記述できず、中身を理解して例外処理を記述したとしてもそのアスペクトの再利用性が低くなってしまう。

そこで、本研究では、分散環境における実験において、状況に応じて適切な例外処理を行なえるツール (AOSted) を提案する。AOSted では分散環境における実験の処理内容を Java 言語で書かれた独自の API を用いてシェルスクリプトを書くように記述できる。また、アスペクト指向プログ

ラミング技術を利用して例外処理のみをメインプログラムから分離してまとめて記述することができる。例外処理を分離して記述されたメインプログラムにはプログラムロジックのみが記載されているため、ユーザはそのプログラムを理解しやすく、保守しやすい。例外が発生した状況によって例外処理を差し替える事が容易になり、プログラムの再利用性が高まる。AOSted には分散環境における実験を行なう際に必要な基本的な処理が簡単に書けるライブラリが提供されているが、AspectJ とは違い、そのライブラリのソースコードを理解しなくても例外処理を容易に記述することができる。また、AOSted では、例外処理の内容を記述する際、例外が発生したホスト名や発生した例外の種類、実行中のクラス名などの発生した例外に関する様々な情報をユーザがに取得することができるためユーザはこれらの情報を利用する事で処理内容を柔軟かつ容易に記述する事ができるようになる。例えば、指定したホストからの応答がなかった場合、代替可能なホストが他に存在するならば、指定するホストを代替ホストに変更して再試行することができる。

謝辞

本研究を進めるに当たり、研究の方針や進め方について数々の助言をしてくださった指導教員の千葉滋助教授に感謝致します。

また、研究の方針について様々な助言をしていただいた光来健一助手に大変感謝致します。同研究室所属の佐藤芳樹氏、東京工業大学大学院の西澤無我氏、松沼正浩氏、柳澤佳里氏、日比野秀章氏には、多くの疑問を聞いて頂き、様々な助言を頂きました。感謝致します。

また、本研究に必要な多くの知識を与えてくださった東京工業大学大学院の須永豊氏、石川零氏、薄井義行氏に感謝致します。

目 次

図 目 次

表 目 次

第1章　はじめに

ネットワーク利用者の増加に伴い、サーバマシンが過負荷状態になることも珍しくない。そのような状態に陥った場合、サーバマシンがダウンしてしまうことを回避するための研究がされている。例えば、スケジューラを改善して過負荷時のサーバの性能を向上させるという研究が行なわれている [2]。このような研究を行なうときに、擬似的に複数のマシンにそれぞれ決められた処理をさせ、サーバマシンに負荷をかけさせるという状況を作り出してテストされる。このテストを分散環境における実験という。

このような分散環境における実験では例外処理が重要である。なぜなら、分散環境では複数のマシンを利用するため、一台のマシンで実験を行う時と比べて、マシンの故障やネットワーク切断などの例外が発生しやすくなるからである。例えば、マシンの台数が揃っている事を前提とした実験では、一台のマシンが故障していた場合、実験に多大な時間を費やしたにも関わらず得られた結果は意味のないものとなってしまう。例えば、負荷のかけ方を変えて実験する場合、想定された負荷をかけていることが前提となっている実験などがある [3]。このような実験で例外が発生したときには状況を修復することができるならば代替可能な他のマシンに置き換えるなどして自動的に修復して実験を続けたい。

しかし、状況に応じて適切に例外処理を施すのは困難である。というのは、同じ例外が発生した場合でも、発生した状況によって、例外処理の内容が変わってくるからである。例えば、発生した例外は同じであっても、異なるホストで例外が発生した場合、施したい例外処理も異なってしまうということである。また、同一ホストで異なる例外が発生した場合でも施したい処理の内容は変わる。シェルスクリプトの内部や Java 言語の `try-catch` 文で例外処理を記述する際、起こりうる例外を細かく指定して、その部分に処理内容を逐一記述すると、例外処理記述がプログラムロジックを記述したメインプログラムのあちこちに散在してしまうことになる。その結果、メインプログラムが見づらくなる他、処理内容を変更した場合には変更しづらくなってしまう。それとは逆に、例外処理をまとめて書こうとすると、細かく制御を指定するのが困難になってしまう。シェルスクリプトにおいては終了ステータスは単なる整数であり、例外に関する情報が少ないため効果的な修復が難しい。Java 言語を拡張した汎用的な

アスペクト指向言語 AspectJ を用いると例外処理記述をメインプログラムから分離して記述することができ、細かく制御することも可能だが、例外処理はメインプログラムに強く依存してしまう。そのため、メインプログラムの中身を理解しないと例外処理が記述できず、中身を理解して例外処理を記述したとしてもそのアスペクトの再利用性が低くなってしまう。

そこで、本研究では、分散環境における実験において、状況に応じて適切な例外処理を行なえるツール (AOSted) を提案する。AOSted では分散環境における実験の処理内容を Java 言語で書かれた独自の API を用いてシェルスクリプトを書くように記述できる。また、アスペクト指向プログラミング技術を利用して例外処理のみをメインプログラムから分離してまとめて記述することができる。例外処理を分離して記述されたメインプログラムにはプログラムロジックのみが記載されているため、ユーザはそのプログラムを理解しやすく、保守しやすい。例外が発生した状況によって例外処理を差し替える事が容易になり、プログラムの再利用性が高まる。AOSted には分散環境における実験を行なう際に必要な基本的な処理が簡単に書けるライブラリが提供されているが、AspectJ とは違い、そのライブラリのソースコードを理解しなくても例外処理を容易に記述することができる。また、AOSted では、例外処理の内容を記述する際、例外が発生したホスト名や発生した例外の種類、実行中のクラス名などの発生した例外に関する様々な情報をユーザがに取得することができるためユーザはこれらの情報を利用する事で処理内容を柔軟かつ容易に記述する事ができるようになる。例えば、指定したホストからの応答がなかった場合、代替可能なホストが他に存在するならば、指定するホストを代替ホストに変更して再試行することができる。

本稿の残りは、次のような構成からなっている。第 2 章は 分散処理実験での例外処理の重要性と従来手法の問題点について述べる。第 3 章では、提案する AOSted の特徴を、第 4 章では、AOSted のこれまでの実装、第 5 章では、AOSted を利用した実験の例、第 6 章では、関連研究を、そして第 7 章でまとめを述べる。付録では、AOSted を使用するにあたって必要な事柄について述べる。

第2章　例外処理の重要性

例外処理は重要なのはプログラムを書いている人なら誰でも一度は感じたことはあるだろう。この章では分散処理における実験を行なう際の例外処理について述べ、従来手法での例外処理の難しさについても述べる。

2.1　分散処理実験について

分散処理における実験は一度の実験で同時に複数のマシンを使用したりするため、一台のマシンで実験を行なう場合より問題が発生しやすくなる。ストレステスト[1]など、サーバ用のハードウェアをテストしているにも関わらず、テストしている環境が想定された条件を満たしていないで、それに気づかずテストを続けてしまった場合、テストの結果は意図したものと違ったり、調査したつもりのものとは異なってしまう。このような場合、環境は必ず整えることができれば便利である。それはつまり、例外が生じた場合の適切な処理をすることで実現できる。

2.2　従来手法の問題点

分散処理における実験で、従来よりよく用いられているシェルスクリプトでの例外処理の問題点を述べる。また、シェルスクリプトでの例外処理の問題点を解決することができる AspectJ での例外処理の仕方と、それの問題点についても述べる。

[1] ハードウェアやソフトウェアに短時間に大量のデータを与えるなどの高い負荷をかけて、製品が正常に機能しているか調べるテスト。ハードウェアやソフトウェアの不良の中には、低い負荷で動作しているときには不具合を起こさないが、高い負荷で動作させると不具合を起こすという性質を持ったものがある。また、不具合を起こす確率が非常に低い不良も含まれていることもあり、通常の使用を想定したテストではなかなか不良が発見できないことがある。このような、不具合を起こす状況が限定された製品の不良を検知するために行われるのがストレステストで、絶対にトラブルを起こすことが許されないサーバ用のハードウェアなど、信頼性の高いシステムに使われる製品に対してよく行われている。

2.2.1　シェルスクリプト

分散処理における実験に限らず、計算機にある決まった処理をさせる場合、よく用いられるのがシェルスクリプトである。シェルスクリプトではシェルごとに独自の文法が採用されているが、おおむね 1 行が 1 つのコマンドとして扱われる。特に UNIX 系 OS のシェルは複雑な繰り返し処理や条件分岐などに対応しており、シェルスクリプトだけでかなり複雑な処理を自動化できる。また、各コマンドではコマンドが正常に実行できたか否かなどの情報を戻り値として返すことができる。この戻り値のことを終了ステータスという。この終了ステータスを用いて正常に終了できなかった場合に対する処理を記述することが可能である。シェルスクリプトで記述した際の例を示す。

シェルスクリプトで記述した時の例

```
#!/bin/bash

hostname=$1
ping ${hostname}          ←プログラムロジック

if [ $? = 1 ]; then       ←例外処理記述
  case ${hostname} in
  "node03")
  hostname=node04
  ;;
  case ${hostname} in
  "node08")
  hostname=node09
  esac
  :
  :
```

シェルスクリプトで書いた場合、プログラムロジックと例外処理内容が混在してしまうのである。

2.2.2　AspectJ

シェルスクリプトで例外処理を記述する際、プログラムロジックと例外処理記述が混在してしまうことを先程述べた。そこでこの節では、プログラムロジックと例外処理記述を分離して記述することを実現するアスペクト指向についてと、汎用てきなアスペクト指向言語 AspectJ で記述した

際、どのような問題が生じるかを述べる。

アスペクト指向

オブジェクト指向とは、プログラムをオブジェクト (もの) としてモジュール化するプログラミング技法であり、それぞれのオブジェクトはある機能を表している。オブジェクト指向は様々な機能をカプセル化する能力に優れており、保守性、拡張性、可読性の高いプログラムを記述できるといわれている。

しかしながら、ある種の機能のための処理群は複数オブジェクトに散らばってしまうことがある。これらの処理群は**横断的関心事** (**crosscutting-concern**) と呼ばれ、上手にオブジェクトにモジュール化できないことが問題視されている。これらの機能をを修正するためには、プログラムの多くの個所を修正する必要が出てしまい、その結果一つの機能が一つのプログラムで表現されるという簡潔性が損なわれ、質の低いプログラムになってしまう。**アスペクト指向プログラミング** (**AOP**) は、この問題を解決するためのプログラミング技法である。AOP 言語では複数のオブジェクトに散らばってしまう機能を、オブジェクトとは別の側面から考慮し、それをアスペクトとしてモジュール化して扱える。特にアスペクトはシステマティックな機能を扱うことが多い。それらにはログ処理、同期制御、永続性、例外処理、トランザクションなどがある。AOP にはこのように、基本モジュールであるオブジェクト (クラス) と、別の側面からの機能を持つアスペクトの二種類のモジュールが存在する。これら別個の意味合いを持つものを合成し、両方の機能を持つモジュールにすることを **weave** といい、その処理系を **weaver** という。この処理によりアスペクトとして分離してモジュール化されていた処理群を、複数のオブジェクトに同時に組み込むことができる。アスペクトには weaver の制御に関するコードが記述されており、具体的にはオブジェクト内のどの個所にどういったコードを合成するかについてが記述されている。

AspectJ のプログラミングモデル

AspectJ は標準的な汎用 AOP 言語の一つであり、Java にアスペクト指向を取り入れて拡張したものである。AOP 言語は数多く提案されているが、その中で AspectJ は代表的なものであり、現在もっとも幅広く使われている。そのため AOP 独自の概念を説明する際には、AspectJ の用語を使うことが多い。この節では AspectJ のプログラミングモデルを説明するとともに、AOP を理解するのに必要な概念を説明する。

- pointcut
  joinpoint とは、プログラム内の演算で、AOP 言語がコードを合成しうる個所のことである。AspectJ にはメソッド呼び出し、フィールド参照、インスタンス生成などがある。joinpoint の種類は言語によって異なる。joinpoint を抽出する作業のことを *pointcut* といい、コードを合成したい個所を具体的に特定するものである。AOP では、アスペクトはオブジェクトに合成されるが、pointcut はオブジェクト内の個所を weaver に支持する。その個所で何を実行するかに関しては、pointcut は関与しない。これは抽出の対象とする joinpoint の種類と、そこに課す条件の対で表せる。

  例えば「`int` 型の返り値で名前が `setX`、引数を持たない `Point` クラスで宣言されている」という「メソッド呼び出し」を抽出したい場合、AspectJ では以下のように記述する。

  ```
  call(int Point.getX())
  ```

  `call` は pointcut 指定子と呼ばれるものであり、メソッド呼び出し joinpoint を抽出の対象としている。括弧内に条件を記述することにより、プログラム内のどのメソッド呼び出しを抽出するかを特定している。

  この他にも AspectJ には数多くの pointcut 指定子が存在する。

  call 特定したメソッド呼び出しを抽出

  set 特定したフィールド読み込みを抽出

  get 特定したフィールド書き込みを抽出

  initialization 特定したクラスのインスタンス生成を抽出

  execution 特定したメソッド実行を抽出

  handler 特定した例外のハンドラを抽出

  staticinitialization 特定したクラスの静的初期化子の実行を抽出

  AspectJ の pointcut 指定子で抽出できる演算は、以上の 7 種類である。このうち上からの 6 つまでの指定子が対象としている演算を AspectJ における基本 joinpoint と呼ぶことにする。これらの pointcut 指定子は joinpoint の種類を対象としていたが、joinpoint の位置を対象として weave 個所を特定する指定子もある。

  within 特定したクラス内の全ての基本 joinpoint、またはそのクラスの静的初期化子

withincode 特定したメソッド内の全ての基本 joinpoint また、プログラムの実行時の情報を基にした pointcut も存在する。以下の 3 種類がそれに相当する。これらの指定子は、アドバイスという機能において

joinpoint の情報を受け渡すためにも使われる。詳細は後述する。
target 特定したクラスのオブジェクトをターゲットとした全ての基本 joinpoint

this 特定したクラスのオブジェクト内で実行した全ての基本 joinpoint

args 特定した引数で実行する全ての基本 joinpoint さらに AspectJ には、プログラムのフローを意識した pointcut 指定子もある。また、`if` 指定子は任意の式を条件として pointcut をすることができる。以下の 3 種類も実行時の情報を基にする指定子である。

cflow 特定した pointcut のフローの中にある全ての基本 joinpoint

cflowbelow 特定した pointcut のフローの中にある全ての基本 joinpoint、ただし pointcut で指定された最初の演算は含まない

if 任意の式を含む全ての基本 joinpoint

pointcut 指定子には複数の joinpoint を一度に抽出できるようにワイルドカードを使えるものがある。以下がワイルドカードを使った例である。

```
get(* Point.*) call(public void Point.set*(..))
```

1 つ目は `get` 指定子を使っておりフィールド読み込みを抽出する。条件は `Point` クラスで宣言される任意の型で任意の名前のフィールドである。2 つ目は `call` 指定子を使っておりメソッド呼び出し読み込みを抽出する。条件は `Point` クラスで宣言される `void` 型で修飾子が `public`、名前が `set` で始まり引数は任意のメソッドである。引数部の `[..]` は、引数の型も数も任意でよいという意味である。

また pointcut 指定子は論理演算子で組み合わせることができ、&&(論理積)、||(論理和)、!(否定) の 3 種類の論理演算子がある。これは例えば、

```
call(public void Point.set*(..))  && !within(Point)
```

のように使う。先ほどと同じように `Point` クラスで宣言されているメソッド呼び出しを抽出するが、`Point` クラス内で起きた演算は抽出しない。

- アドバイス
  pointcut で抽出された複数または単数の joinpoint において、新たなコード断片を実行することを指示する機能をアドバイスという。この際実行されるコード断片をアドバイスボディと呼ぶ。アドバイスには before, after, around の 3 種類があり、ボディを実行するタイミングが異なる。それぞれ pointcut された joinpoint の前、後、もしくはその代わりに実行する。アドバイスのプログラム例は、

```
before() : call(int Point.getX()) {
    System.out.println("before call getX.");
}

before(int newx)
    : call(void Point.setX(int)) && args (newx){
    System.out.println("setting x in Point. " +
                       "new x = " + newx);
}
```

  のようになる。1 つ目は Point クラスの getX メソッド呼び出しの前に、メッセージを出力することを指示する。まず、before の部分でこのアドバイスの実行タイミングを指示する。次にコロンに続くのが pointcut 部であり、プログラム内のどの個所でアドバイスを実行するかを指示する。最後に中括弧で囲まれた部分がアドバイスボディであり、何を実行するかを指示する。アドバイスボディには任意の Java 言語の式を記述できる。

  2 つ**コンテクスト引き渡し**という機能を使っている。これはアドバイスボディの中で、joinpoint の実行時の情報を参照するためのものである。まずメソッドの仮引数を定義するように before(int newx) と記述する。そして args を使い、joinpoint のどの情報が newx に束縛されるかを指示する。この場合は call 指定子とともに使っており、メソッド呼び出しの第一番目の実引数値が束縛される。こうして宣言した newx という変数をアドバイスボディ中で使うことができる。AspectJ ではこの他に target と this 指定子のこの機能がある。call の条件であるメソッドの引数の数と args の引数が異なっている場合は、どのメソッド呼び出しも抽出されない。

- インタータイプ宣言
  既存クラスに新たな要素を追加することをインタータイプ宣言という。追加できる要素には、フィールド、メソッド、コンストラクタ

がある。また既存クラスの階層構造を変えることをインタータイプ宣言といい、親クラスの変更とインタフェースの追加ができる。これは従来イントロダクションと呼ばれていたものと同等である。以下に具体例を示す。

```
private int Point.z = 5;
public int Point.getZ() {
    return z;
}
declare parents : Point implements java.io.Serializable;
declare parents :
    (Point || Line || Rectangle) implements Comparable;
private Strnig Figure+.name = "Figure";
```

1 つ目は、修飾子が `private` で `int` 型の `z` という初期値が 5 のフィールドを `Point` クラスに追加する。2 つ目は、修飾子が `public` で返り値が `int` 型の、引数を持たない `getZ` という名前のメソッドを `Point` クラスに追加する。メソッドボディは中括弧でくくられている部分である。3 つ目はクラスの階層構造を変えるもので、`Point` クラスが `java.io.Serializable` インタフェースを継承するようにする。4 つ目と 5 つ目は、複数クラスを同時にインタータイプ宣言の対象としている。4 つ目は`||` で連結された 3 つのクラスが `Comparable` インタフェースを継承するようにし、5 つ目は `Figure` のサブクラス全てに `name` というフィールドを追加する。AspectJ の使用では明確に述べられていないが、インタータイプ宣言もまた、pointcut とインタータイプボディの対であるといえる。上記の例のインタータイプ宣言対象は、上から 3 つでは `Point` クラスのみであるが、4 つ目と 5 つ目では複数クラスを対象にしている。4 つ目では `Point`, `Line`, `Rectangle` の 3 クラスが対象であり、それら全てにインタータイプボディを適用する。AspectJ のインタータイプ宣言にはコンテクスト引き渡しの機能がなく、weave の対象となったクラスの情報を参照することができない。また複数クラスを対象としていても、インタータイプボディは同一のものでなければならない。

- アスペクト
  横断的関心事をモジュール化したもののことであり、関連する pointcut とアドバイスとインタータイプ宣言が集まっている。またアスペクト自体のメンバとなるフィールドとメソッドを宣言することもできる。以下のプログラムが AspectJ のアスペクト全体を記述した例である。

```
01 aspect CounterAspect {
02     public static void print(int i) {
03     Sysetm.out.println(i);
04     }
05     int Point.count = 0;
06     pointcut pc() : call(public * Point.*(..));
07     before() : pc() {
08         count++;
09         CounterAspect.print(count);
10     }
11 }
```

1 行目の aspect という宣言でこれがアスペクトの役割を果たすことがわかる。これはオブジェクト指向の class という宣言に類似している。2-4 行目はアスペクトメンバとしてメソッドを宣言している。AspectJ の処理系はアスペクトと同名のクラスを生成し、その中でこのメソッドが定義される。5 行目がインタータイプ宣言であり Point クラスに count というフィールドを追加する。6 行目で pc という名前の pointcut を定義している。pointcut はこのようにアドバイスと独立して定義することも可能である。7-10 行目がアドバイスであり、Point クラスの public メソッド呼び出しの前にアドバイスボディ実行するように指示する。ボディは、count を 1 増やすコードと CounterAspect 内の print メソッドを呼び出す。この例ではアスペクトの weave 対象は Point クラスだけに限られているが、多くのクラスを weave 対象とする場合もある。そのような場合でも、カウントのタイミングや方式の変更時には CounterAspect のみを修正すればよく、weave 対象である Point クラスなどを修正する必要はない。

これらをふまえて、AspectJ を用いて例外処理を記述した例を以下に示す。

AspectJ を用いて例外処理を記述した例

```
01 aspect ExceptionHandling {
02     after(UnknownHostException e):
03         withincode(void Transmitter.run()) && handler(e) {
04         ....
05
06
07     }
08 }
```

ここで生じる問題は、示した例の 3 行目を見ると分かるだろう。Transmitter クラスの run() メソッドを実行中に UnknownHostException が生じた場合、このアドバイスが実行されるのであるが、Transmitter.run() などと書かなければならなくなる。この Transmitter.run() というのはプログラムロジックで実行するメソッドであり、例外処理記述をプログラムロジックと分離して記述したにもかかわらず、強く依存してしまっているのが分かる。これだと、このアスペクトは再利用性がなくなってしまうのである。

第3章　AOStedの提案

この章では、分散環境における実験を円滑に行う事ができる支援ツール (AOSted) を提案する。AOSted には以下のような特徴がある。

3.1　例外処理を分離して記述可能

AOSted を用いると、分散環境における実験において例外が発生した場合、どのような対処を行なうかを実験のメインプログラムから分離して記述できることができるようになる。そうすることで、実験のメインプログラムにはプログラムロジックしか書かれておらず、ユーザはメインプログラム、例外処理記述共に可読性が向上し、保守しやすくなる。また、それぞれの再利用性も高まる。

3.1.1　適切な例外処理

適切な例外処理とは、一概にこれといったものはない。実験内容によって、適切な例外処理というのは変わってくる。そこで、AOSted ではユーザーが例外処理内容を自由に書くことができるようになる。

3.1.2　例外に関する情報の取得

例外が発生した時の情報、例えば、発生した例外のみならず例外が発生したホスト名や例外が発生した時間、例外が発生したクラス名・メソッド等が取得できると、例外を適切に処理することができるようになる。しかし、分離して記述することによってそれらの情報が必要にもかかわらず取得でき名場合が多い。AOSted では、例外が発生した場合、先程述べたような情報を詰め込んだオブジェクトを生成し、そのオブジェクトを例外ハンドリングをする部分、つまりユーザが自由に定義することができるメソッドに引数として渡すことによって、ユーザーはそれらの情報を取得しやすくなる。容易に情報を取得することができたユーザーは、それらの情報を用いて適切な例外処理ができるようになるのである。

3.2　独自の API

AOSted では、分散環境における実験を行なう際、基本的な処理として必要となる作業を、簡単に書くことができるライブラリを併せて提案する。基本的な処理とは以下のようなものが挙げられる。

3.2.1　実験プログラムの配布

分散環境における実験を行なう場合、遠隔ホスト上で動かしたい実験プログラムを、事前に手作業で配布するのはかなり面倒である。使用するホストが多くなれば面倒であるのはもちろんのこと、メイン関数が含まれているクラスファイルのみならず、複数のクラスファイルを配布しなければいけなくなってしまう場合も多い。このような場合に便利なのが Load クラスである。Load クラスはホスト名と、そのホストで利用するポート番号、そのホストで実行したい実験プログラムのメイン関数が含まれるクラス名と必要な場合はその引数を与えことにより、指定したホストに、指定したポート番号を利用して、指定したクラスを実行させるのに必要な `.class` ファイルを `.jar` ファイルにまとめて送信することができる。

3.2.2　配布した実験プログラムを実行

複数のホストに配布した実験プログラムを少ない誤差で一斉にスタートさせることができる。これは分散環境における実験に有用な機能である。

3.2.3　実験結果の回収

複数のホストから指定した名前で保存されている実験結果を各ホストから回収することができる。各ホストに実験結果が散らばっている場合、一つ一つ回収するのは面倒である。メインプログラムを書く際、この API を使用して実験が終われば自動的にすぐに回収するようにしておくと

3.2.4　外部コマンドの実行

UvaScript [1] の Task クラスを継承している。UvaScript とは Perl や Ruby のようなスクリプト言語で書くような処理を Java で書くためのツールキットである。

UvaScript の Task クラスを用いると外部コマンドの実行が容易にできるようになる。単純に外部コマンドを起動するだけなら、

```
new Task("ls -l").run()
```

のようにする。この例では Task クラスのコンストラクタの引数

```
ls -l
```

を外部コマンドとして起動することができる。
　複数のコマンドをパイプでつないで起動することもできる。例えば

```
new Task("ls *.java").pipe("wc").run()
```

は、シェルから

```
ls *.java | wc
```

と起動するのと同じになる。
　標準入力や標準出力をファイルにリダイレクトすることも可能である。

```
new Task("sort").in("input").out("output");
```

は、シェルから

```
sort < input > output
```

と起動するのと同じになる。
　AOSted ではこの Task クラスを継承して、バックグラウンド処理もできるようになった API を提案する。

第4章　これまでの実装

この章では、ツールで用いられている既存の技術や概念を説明をしながら、ツールの実装について説明していく。また、ツールの例外ハンドリングを行なっている仕組みについても述べる。

4.1　クラスローダー

ユーザーが書いた実験のメインプログラムや例外処理記述をコンパイルして実行するにはクラスローダーを用いている。

クラスローダーとは、クラスのロードやリソース（ファイル）の検索を担当するオブジェクトで、`java.lang.ClassLoader`を継承したクラスのインスタンスである。全てのクラスはクラスローダーによってアプリケーションにロードされる事になっており、各クラスは自身をロードしたクラスローダーへの参照を保持している。そのクラスローダーは`java.lang.Class`の`getClassLoader`メソッドで取得することができる。クラスローダはツリー構造をしており、関連する親クラスローダーを１つ所有している。ツリー構造の大元に位置するクラスローダーを**ブートストラップ・クラスローダ**と呼ぶ。これはJava仮想マシンに組み込まれており、Javaアプリケーションの起動時に、最初に読み込まれるものである。ブートストラップ・クラスローダーはJava標準のライブラリや、Java仮想マシンの拡張ディレクトリ`jre/lib/ext`に置かれた`jar`ファイルにあるクラスしかロードできない。ブートストラップ・クラスローダの次に位置しているのが、**システム・クラスローダー**である。システム・クラスローダーはブートストラップ・クラスローダーの唯一の子クラスローダである。システム・クラスローダーは、CLASSPATHからクラスのロードやリソースの検索を行うもので、Javaアプリケーションを実行する上で重要な役割を果たすものである。また独自に作成・設定したクラスローダーは、何も指定しなければシステム・クラスローダーを親として持っている。システム・クラスローダーは、もっとも基本的なクラスローダーといえる。クラスローダーの親子関係は、クラスをロードする上で重要な働きをする。デフォルトでは、子クラスローダーがクラスをロードする際、まず親クラスローダーに処理を依頼（委譲）する。親クラスローダで検索できなかった場合

に、はじめて子クラスローダーで検索を試みる。全てのクラスローダーはシステム・クラスローダーを親（先祖）に持っているので、CLASSPATH にあるクラスは、どこからも検索できることになる。

ところでアプリケーションはメインスレッドも含めて、必ず１つ**コンテクスト・クラスローダー**を持っている。コンテキスト・クラスローダーとは、そのスレッド内でクラスをロードする場合に使用されるクラスローダーのことである。これを使用することにより、特定のスレッドでは、親スレッドで使用できないクラスをロードできるようになる。コンテキスト・クラスローダーは、スレッドを生成する親スレッドの側で設定される。もし設定されない場合には、親スレッドのコンテキスト・クラスローダーが設定される。コンテキスト・クラスローダーの設定や取得は、`java.lang.Thread` の `setContextClassLoader()` や `getContextClassLoader()` メソッドで行なうことができる。なお最初に起動されるメインスレッドのコンテキスト・クラスローダーは、デフォルトでシステム・クラスローダーになる。

4.2　ソケット

複数のマシン間でデータの送受信を行なう場合、ツールでは Java で提供されているソケットに関する様々なクラスを用いて実装している。

4.2.1　クライアントソケット

データは、データグラムと呼ばれるサイズ制限のあるパケットによってインターネット越しに転送される。データグラムは、ヘッダとペイロードと呼ばれる情報そのものが含まれる部分から成る。ヘッダには、パケットの宛先アドレスとポート番号、パケットの送信元アドレスとポート番号、その他信頼性の高い転送を保障するためのさまざまな管理情報が含まれている。ペイロードにはデータそのものが含まれる。ただし、データグラムは最大サイズが決まっているので。ペイロードを複数のパケットに分割し、送信先で再度組み立てることもよくある。送信中に１つまたは複数のパケットが失われたり破壊されたりした場合は、そのパケットを再送信する必要がある。また、ぱけっとが送信した順に受信されないこともよくある。ペイロードのパケットへの分割、ヘッダの作成、着信パケットのヘッダの解析、全てのパケットが受信されたかどうかの管理、といった処理を行うのは大変な作業で、膨大な量の複雑なコードを書く必要がある。

しかし、、Berkeley Unix で考案されたソケットがあるためこれらの処理を行うコードをユーザが自分で書く必要はない。ソケットを使えば、ネットワーク接続を、バイトを読み書きできるもう１つのストリームのように

扱うことができます。ソケットは、Unix で最も重要な考え方を拡張したものである。それは、キーボードであれ、ネットワーク接続であれ、すべての入出力はファイル入出力と同じように扱えなければならないという考え方だ。ソケットを使用すれば、プログラマは、メディアタイプ、パケットサイズ、パケットの再送、ネットワークアドレスといったネットワークの低レベルの詳細部分を意識しなくてすむ。この抽象化の恩恵はきわめて大きいことがわかってきたため、今では最初にそれを実現した Berkeley Unix だけでなく、すべての系列の Unix、さらには Windows、Macintosh、そしてもちろん Java でも採用されている。

ソケットの基礎

ソケットは、2 つのホスト間のコネクションである。次の 7 つの基本動作を実行します。

- リモートマシンとの接続
- データの送信
- データの受信
- 接続の切断
- ポートとの結合
- 着信データの受信待ち
- 結合されたポート上でのりモーとマシンからの接続要求の受け付け

Java の `Socket` クラス（クライアントとサーバの両方で使用される）には。このうち最初の 4 つの動作に対応するメソッドが定義されている。残りの 3 つは、クライアントからの接続要求を待つサーバだけで必要になるため、次章で説明する `ServerSocket` クラスで実装されている。クライアントソケットは通常、以下の順序で使用する。

1. `Socket()` コンストラクタで新しいソケットを作成する。
2. このソケットでリモートホストとの接続を確立する。
3. 接続が確立すると、ローカルおよびリモートのホストが、ソケットからの入力および出力ストリームを取得し、それらのストリームを使用して互いにデータを送信する。この接続は全二重なので、双方が同時にデータを送信および受信できる。データの意味はプロトコ

ルによって異なる。例えば、FTP サーバに送信されるコマンドは HTTP サーバに送信されるコマンドと異なる。ただし、双方が同意したハンドシェイキングの後にデータの送受信を開始するという点は、どのプロトコルでも同じである。

4. データの転送が終了したら、一方または両方のホストが接続を切断する。HTTP1.0 のようにリクエストに対する応答がサーバから返されるたびに接続を切断するプロトコルもあれば、FTP のように 1 回の接続で複数のリクエストを処理できるプロトコルもある。

Socket クラス

`java.net.Socket` クラスは、クライアント側の TCP 操作を実行する Java の基本クラスである。TCP ネットワーク接続を行なう他のクライアント用のクラス（`URL`、`URLConnection`、`Applet`、`JEditorPane` など）はすべて、最終的にこのクラスのメソッドを呼び出している。このクラス自体は、ネイティブの（OS 固有の）コードを使用して、ホストオペレーティングシステムのローカルの TCP プロトコルスタックとやり取りする。`Socket` クラスのメソッドはコネクションのセットアップと切断、およびさまざまなソケットオプションにしたがった設定を行なう。TCP ソケットは信頼性の高いコネクションなので、`Socket` クラスがプログラマに提供するインタフェースはストリームである。ソケットを介した実際のデータの読み書きは、お馴染みのストリームクラスを用いて行なえる。

4.2.2　サーバソケット

前章では、ソケットをクライアント側の視点から説明した。クライアントは、接続町のサーバへソケットをオープンするプログラムである。しかし、クライアントソケットだけでは通信できない。クライアントは通信相手のサーバがいなければ役に立たない。そう考えると、前章で紹介したソケットだけではサーバは書けないことが分かる。`Socket` を作成するには、接続先のインターネットホストを知る必要がある。ところが、サーバを書くプログラマは、だれが接続してくるか前もって知ることはできない。たとえ知ることができたとしても、そのホストがいつ接続してくるのかは分からない。言い換えると、サーバは、電話のそばに座って電話が鳴るのを待っている受付係のようなものである。受付係はだれがいつ電話してくるのか知らない。ただ、電話が鳴ったら、相手がだれであれ受話器を取ってその相手と話すだけである。このような動作は、`Socket` クラスだけではプログラミングできない。もちろん。Java で書かれたクライアントが

Java で書かれたサーバとやり取りしなければならない理由はない。実際、クライアントはさーばの実装言語や実行プラットフォームなどまったく意識しない。

しかし、Java でサーバを書くためのクラス、ServerSocket クラスが用意されている。基本的に、サーバソケットの仕事は、電話の前に座って相手からの呼び出しを待つことである。もう少し技術的にいうと、ServerSocket はサーバ上で動作し、TCP 接続要求を待機する。各 ServerSocket はサーバマシン上の特定のポートで待機する。リモートホストのクライアント Socket がそのポートに接続してくると、サーバは動作を開始し、そのクライアントとの間でコネクションを張るためにネゴシエーションし、2 つのホスト間に（サーバソケットではない）通常の Socket をオープンする。言い換えると、サーバソケットは接続要求を待ち、クライアントソケットが接続要求を送信する。いったんサーバソケットがコネクションをセットアップすると、そのサーバは通常の Socket オブジェクトを使用してクライアントにデータを送信する。データは常に通常のソケット経由でやり取りされる。

ServerSocket クラス

ServerSocket クラスは、Java でサーバを書くために必要なすべての機能を備えている。すなわち、新しい ServerSocket 作成するコンストラクタ、指定されたポートで接続要求を待機するメソッド、コネクションが確立したときデータ送受信用の Socket オブジェクトを返すメソッドが定義されている。これらに加えて、さまざまなオプションを設定するメソッドや、toString() といったその他の通常メソッドをも定義されている。

サーバの基本的なライフサイクルは次の通りである。

1. ServerSocket コンストラクタにより、特定のポート上に新しい ServerSocket が作成される。

2. ServerSocket は、accept() メソッドを使用して手順 1 のポートに対する接続要求を待つ。accept() はクライアントが接続してくるまでブロックする。クライアントが接続してきたら、accept() はそのクライアントとサーバを接続する Socket オブジェクトを返す。

3. サーバのタイプによって、Socket の getInputStream() メソッド、getOutputStream() メソッド、またはその両方が呼び出される。これらのメソッドは、クライアントと通信する入力および出力ストリームを取得する。

4. サーバとクライアントは、接続を切断するまで、互いに同意したプロトコルにしたがってやり取りをする。

5. サーバ、クライアント、または双方が接続を切断する。

6. サーバは手順 2 に戻り、次の接続要求を待つ。

4.2.3　UDP データグラムとソケット

これまでの章では、TCP プロトコルを使用したアプリケーションを書く方法について説明してきた。TCP は、信頼性の高いデータ転送を実現するために設計されたプロトコルである。転送中にデータが失われたり破壊されたりすると、TCP はそのデータを再送信する。パケットが送信した順序で到達しなかった場合、TCP はそれらを正しい順序に並び替える。特定のコネクションでデータが届くのが速すぎる場合は、パケットが失われないように転送速度を落とす。プログラムは、受信したデータが壊れている心配をする必要はない。しかし、この高信頼性を実現するには代価を払わなければならない。代価とはスピードである。TCP 接続を確立したり切断したりするにはかなりの時間がかかる。この接続のオーバヘッドは、サイズの小さいデータを頻繁にやり取りすることが多い HTTP のようなプロトコルでは特に大きくなる。

UDP（User Datagram Packet）は、高速ではあるが信頼性の低い IP 上でのデータ送信を行なう、もう一つのプロトコルである。つまり、UDP でデータを送信した場合、データを構成する各パケットが送信した手順に宛先に到着したかどうかはもとより、そもそもデータが宛先に到着したかどうかさえ知る手段がない。しかし、データが無事に届いた場合、そのスピードは TCP よりずっと速くなる。

UDP プロトコル

では、どうしてこのような信頼性の低いプロトコルを使うのだろうか。送信する価値のあるデータなら、当然、データがきちんと到着したかどうか気にするはずである。当たり前だが、UDP は、潜在的に信頼性の低いネットワーク上で信頼性の高いデータ転送を必要とする FTP のようなアプリケーションには向かない。しかし、すべてのデータを 1 ビットに至るまで正確に転送するより、転送速度のほうが重要視されるアプリケーションもたくさんある。

Java では、UDP を 2 つのクラス、すなわち `DatagramPacket` と `DatagramSocket` により実装している。`DatagramPacket` クラスは、デー

タグラムと呼ばれる UDP パケットの組み立てや受信したデータグラムの分解を行なう。DatagramSocket クラスは、UDP データグラムの送信と受信を行なう。つまり、データを送信するには、データを DatagramPacket として組み立て、そのパケットを DatagramSocket を使用して送信する。データを受信するには、DatagramSocket から DatagramPacket オブジェクトを受信し、パケットの内容を読み取る。ソケット自身はきわめてシンプルである。UDP では、データグラムに関する宛先を含むすべての情報がパケット自身に埋め込まれている。したがって、ソケットは、パケットを待機したり送信したりするローカルポートだけ知っておけばよいのである。

この部分は、TCP で使用する Socket および ServerSocket クラスと対照的である。第 1 に、UDP にはサーバソケットという概念がない。つまり、データを送信するとき使用するソケットと、接続要求を待機するとき使用するソケットを区別しない。第 2 に、TCP では、最終的には送信データをパケット化するものの、TCP ソケットのおかげでネットワーク接続をストリームとして扱うことができる。プログラムは、ソケットから取得した入力ストリームと出力ストリームを使用してデータを送受信する。UDP ではこのようなことができない。プログラムは、データグラムパケットを 1 つ 1 つ処理することになる。1 つのデータグラムに埋め込んだすべてのデータは 1 つのパケットとして送信され、この単位で受信されるか転送途中で失われるかである。個々のパケットは互いに独立しており、2 つのパケットが与えられたとき順番としてどちらが先でどちらが後かを決定する方法はない。ストリームには必須の順序つきデータキューといったものはなく、データグラムはできるだけ早く宛先に到達しようとする。第 3 の相違点は、上記の 2 つの相違点の結果でもあるが、1 つの DatagramSocket が複数のホストとデータを送受信できるという点である。UDP のソケットは、TCP ソケットのように、特定のコネクション専用ではない。というより、UDP にはそもそも 2 つのホスト間のコネクションという概念はないのである。UDP が意識しているのは個々のデータグラムだけである。よって、だれがどのようなデータを送信してきたかを、アプリケーションが自分で解釈しなければならない。

4.2.4　マルチキャストソケット

これまでの章で見てきたソケットは、ポイントツーポイントの通信を実現するユニキャストソケットである。ユニキャストソケットは、2 つの明確な端点間のコネクションを作成する。送信者と受信者はそれぞれ 1 人だけである。いつでも互いの役割を交換できるが、どちらが送信者でどち

らが受信者かは簡単に区別できる。確かに、人類は何千年にも渡って 1 対 1 の会話に従事してきたわけだから、ポイントツーポイントは、ほとんどとはいわないなでも多くのニーズに応える通信形態だが、異なるモデルを必要とする状況も数多く存在する。例えば、テレビ局はある場所から送信機で送ることができる範囲内のすべての場所にデータを送信する。送信されて電波は、受信側のテレビが電源を入れているかどうか、またその曲にチャンネルを合わせているかどうかに関係なく、到達可能な範囲内のすべてのテレビに届く。アンテナの代わりにケーヴるボックスのある家庭にも、テレビのない家庭にも電波は届いている。これがブロードキャスト（同報通信）の典型的な例である。ブロードキャストは、相手を選ばない無差別な方式であり、帯域幅と電力の両方の無駄使いである。

対照的にテレビ会議では、特定のグループに対して音声と画像を送信する。Usenet ニュースは 1 つのサイトに投稿され、世界中の何万という人たちに配信される。DNS ルータはの更新データは、変更をアナウンスしたサイトから他の多くのルータに伝わる。しかし、送信者は中間サイトでメッセージをコピーし、下流サイトに転送してもらう必要がある。送信者は、最終的にメッセージを受信する全てのホストに直接メッセージを送信するわけではない。これらはマルチキャストの例である。マルチキャストは、TCP や UDP 上にアプリケーション層プロトコルを追加することで実装される。これらのプロトコルは、かなり詳細な設定と人間の仲介を必要とする。例えば、Usenet に参加するには、あなたにニュースを送信してくれて、あなたの投稿を他の人たちに転送してくれるサイトを見つける必要がある。ニュース管理者は、あなたを配信先に追加するために、あなたのサイトを config ファイルに追加しなければならない。しかし、最近の主要なオペレーティングシステムのネットワークソフトウェアやインターネットルータによって、各ホストにメッセージを効率的に転送する方法をルータが決定する本当の意味でのマルチキャストを実現できる可能性が開けてきた。つまり、送信先のルータは受信ホストの近くにあるルータにメッセージを 1 コピーだけ送信し、受信側のルータが宛先または宛先近くの複数の受信者に向けて個別にメッセージを送信するという方法である。インターネットのマルチキャストは、UDP 上に構築される。Java では、前章で紹介した `DatagramPacket` クラスと、本章で紹介する `MulticastSocket` クラスを組み合わせてマルチキャストを実装する。

マルチキャストソケットとは

マルチキャストはポイントツーポイントのユニキャスト方式よりは広く、ブロードキャストよりは狭い、相手を絞った方式である。マルチキャ

ストでは 1 つのホストから多くの異なるホストにデータを送信するが、すべてのホストに直接データを送信するわけではない。データは、マルチキャストグループに参加することによってそのデータへの関心を表明したクライアントだけに送信される。これは、ある意味で、公開ミーティングに似ている。公開ミーティングでは、参加者は自由に出入りできる。話題に興味がなくなったらいつでも席を立ってかまわない。ミーティングに参加する前、および退席した後は、情報は一切届かないので何も処理をする必要はない。このような「公開ミーティング」をインターネット上で実装するには、マルチキャストソケットを使用するのが一番である。マルチキャストソケットは、データの 1 コピーを、そのデータへの関心を宣言した当事者に近いロケーションまたはロケーショングループに送信する。

マルチキャストアドレスとマルチキャストグループ

マルチキャストアドレスとは、マルチキャストグループと呼ばれるホストグループのアドレスである。最初にマルチキャストアドレスについて説明する。マルチキャストアドレスには、224.0.0.0 から 239.255.255.255 までの IP アドレスが割り当てられている。この範囲内のアドレスはすべて、先頭 4 ビットが 1110 になる。他の IP アドレス同様、マルチキャストアドレスにもホスト名がある。

マルチキャストグループとは、1 つのマルチキャストアドレスを共有するインターネットホストの集まりである。特定のマルチキャストアドレスに送信されたデータは、そのグループ内のすべてのメンバーホストに転送される。マルチキャストグループとはオープンである。つまり、だれでも自由に出入りできる。グループには永続的なものと一時的なものがある。永続的なグループには、グループ内のメンバーがいるかどうかに関係なく、固定的にアドレスが割り当てられる。しかし、たいていのマルチキャストグループは一時的である。つまり、そのグループ内にいる間だけ存在する。新しいマルチキャストグループでを作成するには、225.0.0.0 から 238.255.255.255 の範囲内から任意のアドレスを選択して、そのアドレスの `InetAddress` オブジェクトを作成したら、後はそれに対してデータを送信するだけである。

クライアントとサーバ

マルチキャストグループにデータを送信するとき、ホストはそのデータをマルチキャストデータグラム（マルチキャストグループ宛の UDP データグラム）に格納する。マルチキャストデータは信頼性が低いものの、コネクション指向の TCP の 3 倍の速度で転送可能な UDP 経由で送信され

る。データ紛失に弱いマルチキャストアプリケーションを開発する場合は、転送中にデータが破壊されたかどうか、失われたデータをどのように処理するかをアプリケーションの責任で判断する必要がある。

Java では、`java.net.MulticastSocket` クラスを使用してマルチキャストする。`MulticastSocket` クラスは、`java.net.DatagramSocket` のサブクラスである。ツールの中で、配布したプログラムをスタートさせる部分にこのマルチキャストソケットを使用している。実際のプログラムは以下のようになっている。

送信側

```
static int ePort;
static String multiAddress = "224.0.0.1";

try{
    // ソケットを作成する
    MulticastSocket mSocket = new MulticastSocket();

    InetAddress group
    = InetAddress.getByName(multiAddress);

    // 送信するパケットを作成する
    byte[] bytes = new byte[] {1};

    // データグラムパケットを作成する
    DatagramPacket packet =
    new DatagramPacket(bytes, bytes.length, group, ePort);

    // パケットを送信する
    mSocket.send(packet);

    // データを送信し終わったのでソケットをクローズする
    mSocket.close();
    }
    catch (UnknownHostException e){
        e.printStackTrace();
    }
    catch (IOException e){
        e.printStackTrace();
    }
}
```

受信側

```
Socket s;
ServerSocket socket;
static String ipAddress = "224.0.0.1";
int port = ePort;

OutputStream os = s.getOutputStream();
int port = socket.getLocalPort();

// マルチキャストアドレスの割り当て
String group = ipAddress;

// マルチキャストソケットを作成し
// それをポート番号'port' にバインド
MulticastSocket ms = new MulticastSocket(port);

// マルチキャストグループに参加する
ms.joinGroup(InetAddress.getByName(group));

// ソケットの準備が完了し、パケットを受け取る準備ができた状態

// データグラムパケットを作成
Create a DatagramPacket and do a receive
byte buf[] = new byte[1];
DatagramPacket pack = new DatagramPacket(buf, buf.length);

// 受け取る
ms.receive(pack);
```

4.3　アスペクト指向プログラミングと例外ハンドリング

AOSted における例外ハンドリングのメカニズムはアスペクト指向の概念の応用である。

制御プログラム

```
String[] hosts = { node01, node02, node03};
int[] port = {3000, 3000, 3000};
String[] program = {Hello, Hello, Hello};
String[] args = {"1 2", "3 4, "5 6"};
Load loader = new Load( hosts, port, program, args);
loader.setErrorHandler(new MyLoadErrorHandler());
loader.load();
…
```

例外ハンドラ

```
class MyLoadErrorHandler extends LoadErrorHandler {
    public void unknownHost(FailEvent event){
        //例外処理記述
        if (e.getHostName().equals("node01")){
            e.replaceHost("node04");
        }
        else if (e.getHostName().equals("node02")){
            e.replaceHost("node05");
        }
        else if (e.getHostName().equals("node03")){
            e.replaceHost("node06");
        }
    }
}
```

AOSted の内部実装

```
public class Load {
    :
    :
    void setErrorHandler(handler){
        ハンドラを登録;
    }
    void load(){
        try{
            :
            :
        }
        catch (UnknownHostException e){
            FailEvent event = new FailEvent( host, e,
…);
            handler.unknownHost(event);
        }
    }
}


class LoadErrorHandler {
    public void unknownHost(FailEvent event){
        //何もしない
    }
}
```

制御プログラムと例外ハンドラについてはユーザが自由に記述することができる。例外処理を記述する際、デフォルトの例外ハンドラクラス継承し、メソッドをオーバーライドすることでユーザが書いたコードを実行することができるようになるのである。

第5章　AOStedの使用例

この章では、AOSted の使用例を順を追って述べる。

1. AOSted を起動すると図 5.1 のような GUI が現れる。

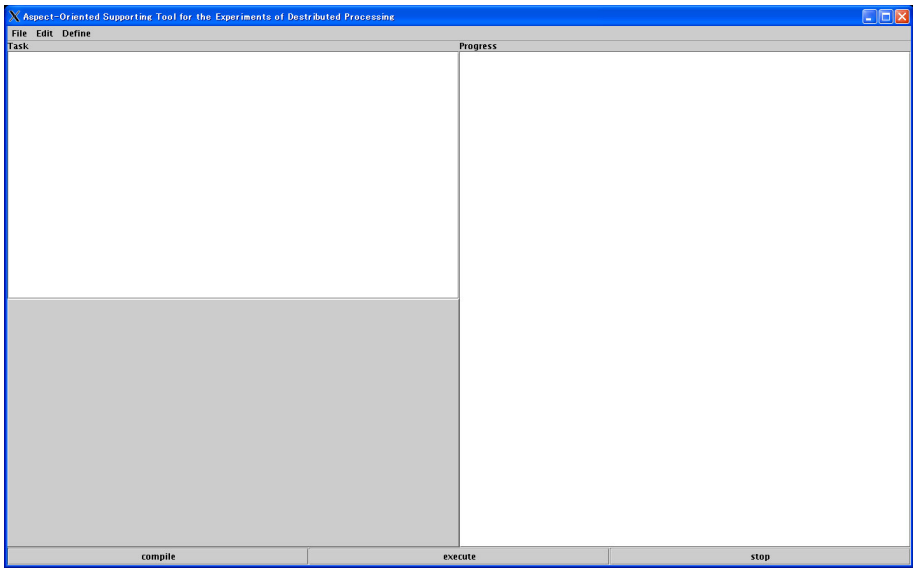

図 5.1: 起動直後の GUI

2. 次に、図 5.2 のように GUI の左上のテキストエリアに実験のメインルーチンを書く。
   ここには `if` 文はもちろんのこと、`for` 文や `while` 文も書くことができるが、クラスやメソッドを定義することはできない。その代わりに、AOSted では分散環境における実験で基本的に必要となる作業に関しては独自の API を用意している。それらについては、次章に詳しく述べている。

   また、保存してあるファイルを読み込むことができる。ファイルを読み込む方法を述べる。まず、図 5.3 のようにメニューバーの”File”を選択すると図 5.4 のような FileChooser が現れる。

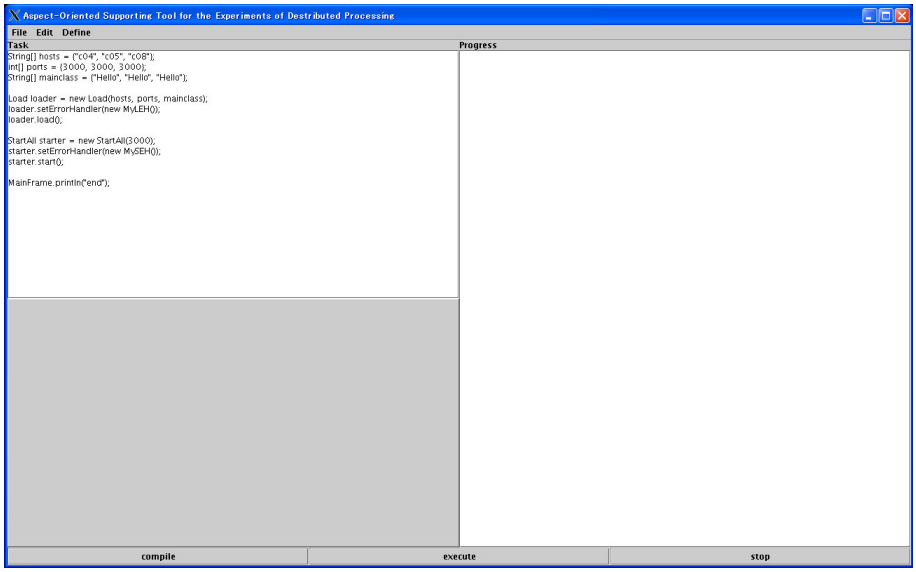

図 5.2: メイン処理を記述

そして、これを使って読み込みたいファイルを選択すると、メインルーチンを書くテキストエリアに読み込んだファイルの内容が表示される。

3. メインルーチンを入力したら、メインルーチンで指定したエラーハンドラーを定義する。エラーハンドラーを定義するには、まず、図 5.5 のようにメニューバーの”Define”を選択する。その後、ハンドラーのスーパークラスになるクラス名を選択する。

4. すると次に図 5.6 のようなウインドウが現れる。

   ここには、先程選択したクラスのサブクラスになるクラス名を入力する。入力し終えると Enter（Return）を押せばよい。

5. すると、図 5.7 のようにメインの GUI の左下にタブつきのテキストエリアが現れる。

   タブには先程入力したサブクラスの名前が表示されており、テキストエリアには

   ```
   public class <サブクラス名> extends <スーパークラス名>{

   }
   ```

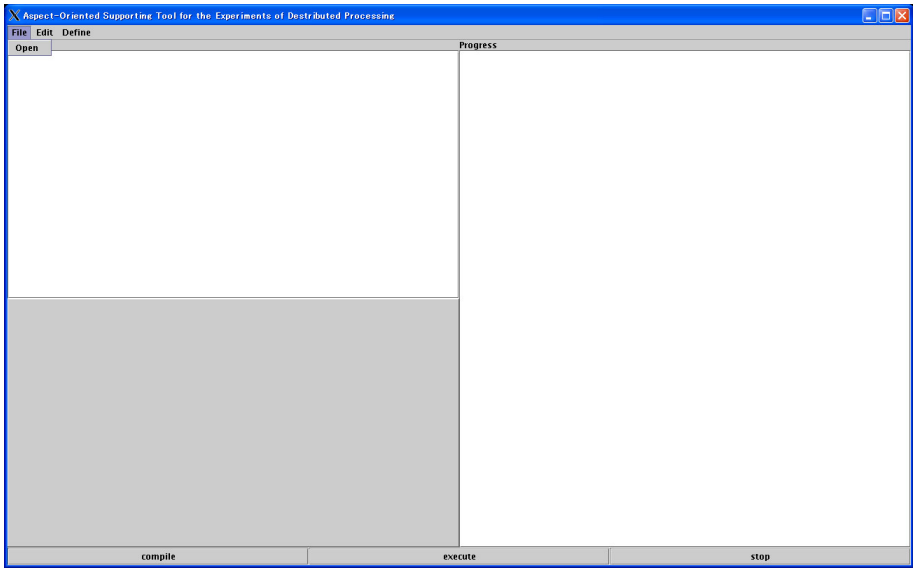

図 5.3: メニューの”File”を選択

が既に入力されている。ここに、図 5.8 のようにスーパクラスのメソッドをオーバーライドして、例外が発生した時に行ないたい処理を書く。
ここには、先程のメインプログラムを書く部分とは違ってメソッドやクラスを定義することができる。また、どのメソッドにも引数として渡されている FailEvent オブジェクトと、予め用意された関数を用いて柔軟かつ容易に例外処理を記述することが可能である。

6. 例外ハンドラを複数定義するには上の作業を繰り返す。

7. メイン処理と例外処理の両方を記述したら、それらをコンパイルする。コンパイルするには GUI の右下の”compile”ボタンを一度押す。
   コンパイルに失敗すると図 5.10 のように GUI の右側にエラー文が表示されるので、修正して再度コンパイルする。

8. コンパイルできたら実行ボタンを押す。すると図 5.11 のようにメイン処理が実行される。

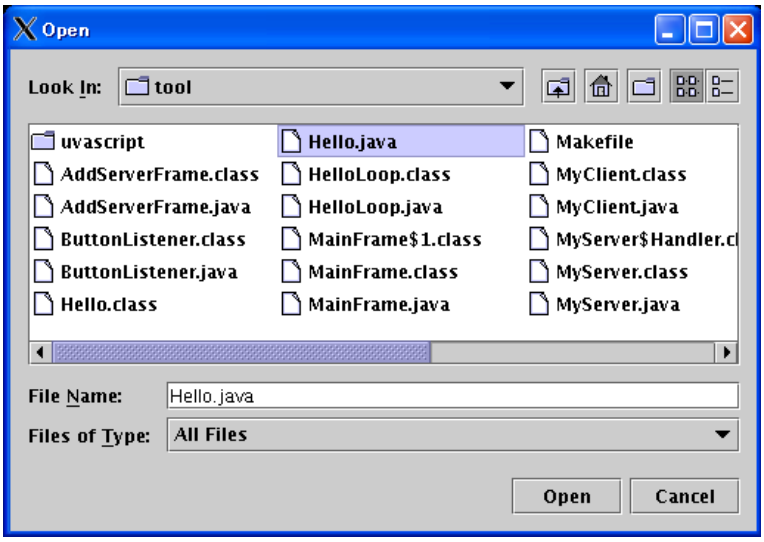

図 5.4: メニューの”Open”を選択後表示される JFileChooser

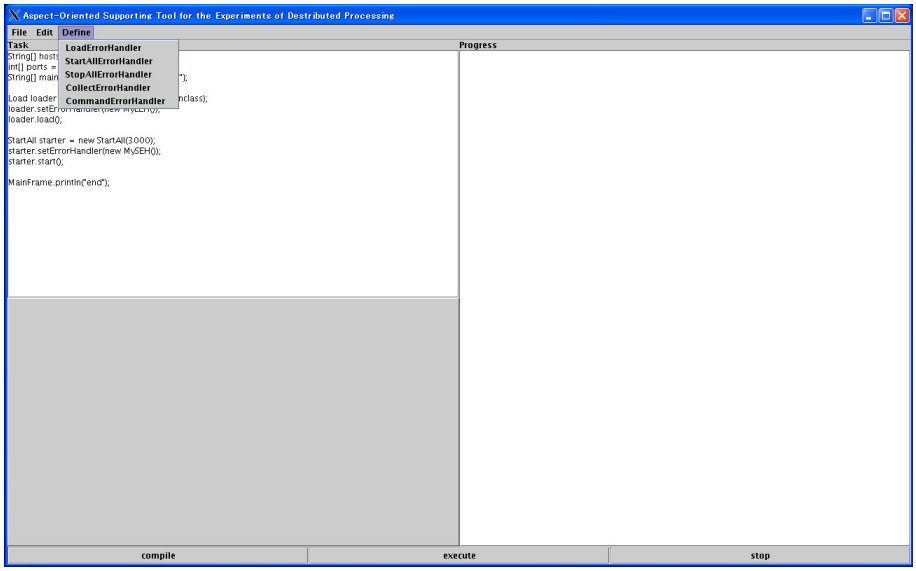

図 5.5: メニューの”Define”を選択

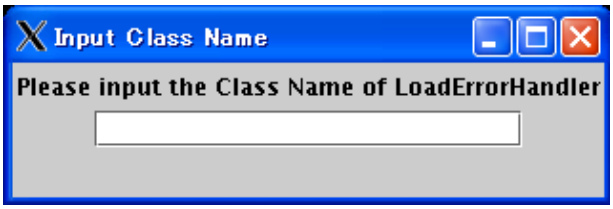

図 5.6: ハンドラークラスの名前の入力要求

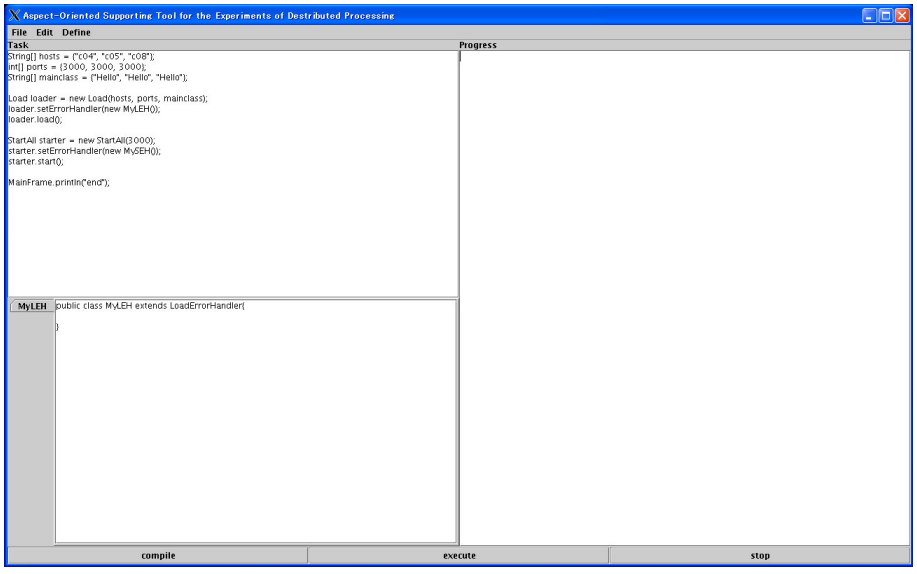

図 5.7: 名前を決定したあと

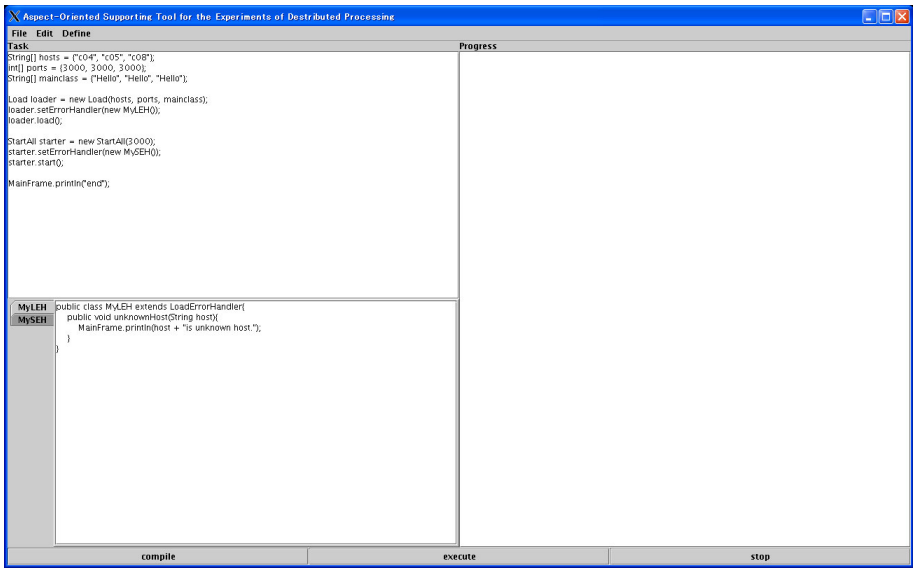

図 5.8: 例外処理内容の記述

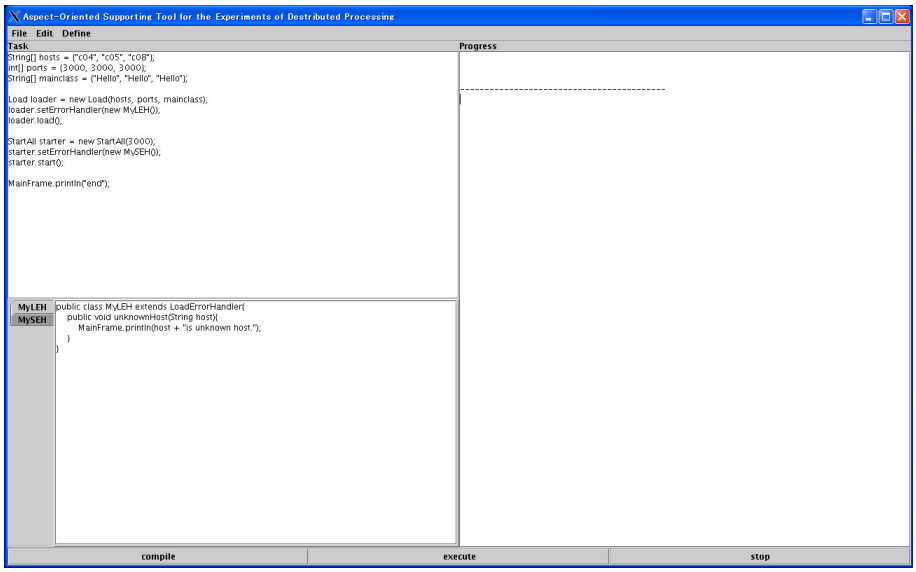

図 5.9: コンパイル成功時

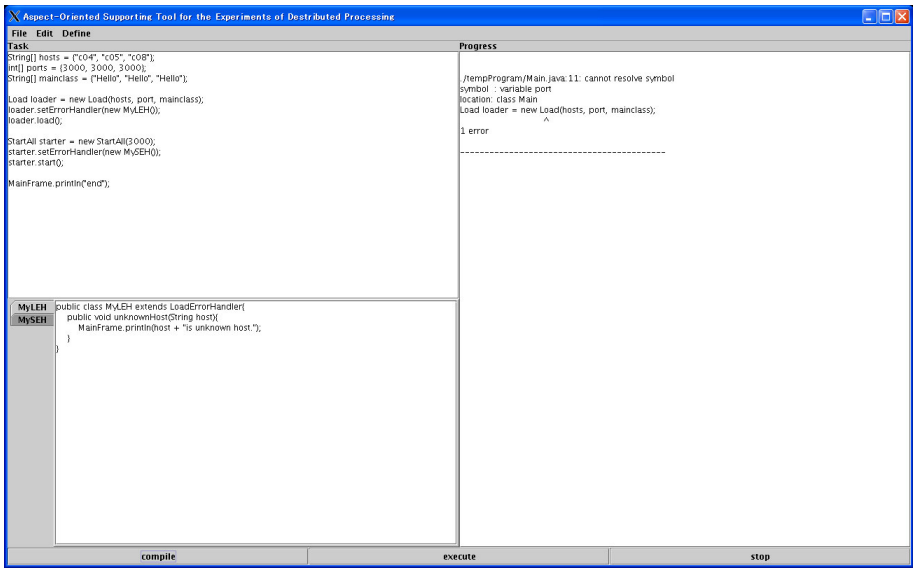

図 5.10: コンパイルエラー画面

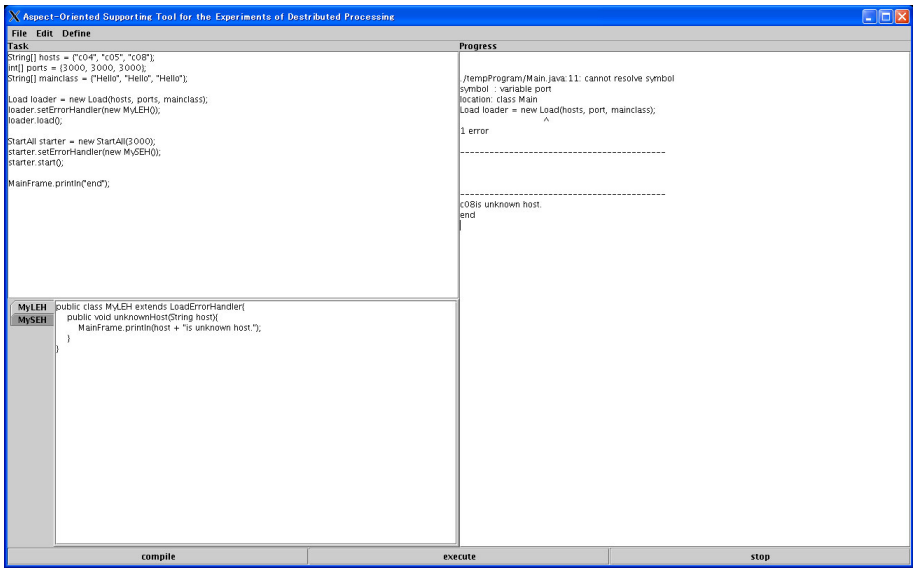

図 5.11: 実行

第6章　実験

以下のようなプログラムを実行したとき、あるホストからの応答がなかった場合、代替可能な異なるホストで実行させることができたことを確認した。

制御プログラム

```
String[] hosts = {"c03", "c05", "c08"};
int[] ports = {3000, 3000, 3000};
String[] mainclass = {"Hello", "Hello", "Hello"};

Load loader = new Load(hosts, ports, mainclass);
loader.setErrorHandler(new MyLoadErrorHandler());
loader.load();

StartAll starter = new StartAll(3000);
starter.start();

...
```

例外処理プログラム

```
public class MyLoadErrorHandler extends LoadErrorHandler {
    public void unknownHost(){
        if (e.getHostName().equals("c03")){
            e.replaceHost("c04");
        }
        else if (e.getHostName().equals("c05")){
            e.replaceHost("c06");
        }
        else if (e.getHostName().equals("c08")){
            e.replaceHost("c09");
        }
    }
}
```

Hello.java

```
public class Hello {
    public static void main (String args[]) throws Exception{
        new HelloLoop();
    }
}
```

HelloLoop.java

```
public class HelloLoop{
    public HelloLoop(){
        while(true){}
    }
}
```

始めに指定したホスト c03, c05, c08 のうち、応答があったホストは c05 のみで、c05 ではプログラムが正常に起動していた。残りの c03 と c08 に関してはホストからの応答がなく、例外ハンドラに記述されているように代わりのホスト c04 と c09 で正常にプログラムが起動していることを確認した。

参考文献

[1] 千葉滋: UvaScript の使い方, http://www.csg.is.titech.ac.jp/~chiba/uva/index.html.

[2] 松沼正浩, 日比野秀章, 佐藤芳樹, 光来健一, 千葉滋: 過負荷時の Web アプリケーションの性能劣化を改善する Session-level Queue Scheduling (2005).

[3] 日比野秀章, 松沼正浩, 光来健一, 千葉滋: アクセス集中時の Web サーバの性能に対する OS の影響 (2004).

付 録A　AOStedを利用するにあたって

AOSted を利用するにあたって、予め設定しておくべきことがある。それを以下に述べる。

A.1　.rhostsの設定

AOSted では、遠隔ホストで Java が起動していない状態から実験を始めることができる。というのは、はじめに rlogin コマンドを実行してホストにログインし、ソケットが張られるのを待つ Java のプログラムを送り、そのプログラムを各ホストで実行させる部分から始めることができるようになっているからである。しかし、rlogin コマンドを実行した場合、毎回パスワードを尋ねられ、毎回パスワードを打ち込むのは面倒である。そこで、予めあるファイルを用意しておくとその状況を回避できるようになる。

あるファイルとは、.rhosts という名前のファイルである。このファイルを作成すると、rlogin コマンドを実行して他のホストにログインする際のパスワード入力を省略できるようになるのである。

遠隔ホストの自分のホームディレクトリの下に.rhosts ファイルを作り、その中にアクセスを許可するサーバ名とその遠隔ホストでのログイン名を記述しておくことによって、その遠隔ホストに対してはパスワードの入力を行うことなくリモートログインできるようになる。.rhosts ファイルの例を次に示す。

```
c04 kumahara
c05 kumahara
s2 kumahara
```

1 行にホスト名とユーザ名を記述する。（できればホスト名だけの行とドメイン名を付加した行との両方を記述するほうがよい。）また、保護属性は 600 あるいは 400 にすることを忘れないようにする。

```
% chmod 600 .rhosts
```

付 録B　AOStedで使用できるAPIとその仕様

AOSted では分散環境における実験において必要な基本的機能を API として提供している。それらを以下の表に示す。これらを使用することによって、制御プログラムを簡単に書くことができるようになる。また、挙げた基本的機能を実行している時に起きたエラーイベントをハンドリングするクラスと、例外が生じた場合に生成されるエラーイベントオブジェクトについて述べる。

インタフェース AostedImpl	
メソッドの概要	
public void	setErrorHandler(AostedErrorHandler handler) エラーイベントハンドラーを登録する
既知の実装クラスの一覧	
Load, StartAll, StopAll, CollectResult, Command	

表 B.1:　AOSted で使用できる API とその仕様（AOStedImple インタフェース）

クラス Load	
フィールドの概要	
public String[]	hosts Jar ファイルを送信するホスト名が格納されている String 型の配列
public int[]	port 送信するポート番号が格納された int 型の配列
public String[]	mainClass 遠隔ホストで実行したいプログラムの メイン関数を含むクラスファイル名を 格納した String 型の配列
public String[]	argv 遠隔ホストで実行するプログラムにおける引数を 格納した String 型の配列
public LoadErrorHandler	handler エラーイベントハンドラ
コンストラクタの概要	
public	Load(String[] host, int[] port, String[] mainClass) 指定されたホスト名に指定されたポート番号を使って、 指定されたメイン関数を含むクラスファイルを実行するための Jar ファイルを送信する新しい Load インスタンスを生成する
public	Load(String[] host, int[] port, String[] mainClass, String[] args) 指定されたホスト名に指定されたポート番号を使って、 指定されたメイン関数を含むクラスファイルを実行するための Jar ファイルを送信し、 そのファイルを各ホストで指定された引数で 実行する準備をさせる新しい Load インスタンスを生成する
メソッドの概要	
void	setErrorHandler(LoadErrorHandler handler) エラーイベントハンドラーを登録する
void	load() 送信を開始する

表 B.2: AOSted で使用できる API とその仕様（Load クラス）

クラス StartAll	
フィールドの概要	
public int	port Load で送信したプログラムを開始させる 合図を送信するポート番号
public StartAllErrorHandler	handler エラーイベントハンドラー
コンストラクタの概要	
public	StartAll(int port) Load で送信したプログラムを開始させる合図を 指定されたポート番号で送信する 新しい StartAll インスタンスを生成する
メソッドの概要	
public void	setErrorHandler(StartAllErrorHandler handler) エラーイベントハンドラーを登録する
public void	start() 開始の合図を送信する

クラス StopAll	
フィールドの概要	
public StopAllErrorHandler	handler エラーイベントハンドラー
コンストラクタの概要	
public	StopAll(int port) StartAll で開始させたプログラムを終了させる 合図を送信するポート番号
メソッドの概要	
public void	setErrorHandler(StopAllErrorHandler handler) エラーイベントハンドラーを登録する
public void	stop() 終了の合図を送信する

表 B.3: AOSted で使用できる API とその仕様（基本的機能 1 ）

クラス CollectResult	
フィールドの概要	
public CollectErrorHandler	handler エラーイベントハンドラー
コンストラクタの概要	
public	CollectResult(String[] host, String[] fileName, String[] saveName) 指定されたホストから指定されたファイルを回収し、 指定されたファイル名で保存する 新しい CorrectResult インスタンスを生成する
メソッドの概要	
public void	setErrorHandler(CollectErrorHandler handler) エラーイベントハンドラーを登録する
public void	collect() 回収を開始する

クラス Command	
フィールドの概要	
public CommandErrorHandler	handler エラーイベントハンドラー
コンストラクタの概要	
public	Command(String cmd) 指定されたコマンドを実行するための新しいプロセスを生成する
public	Command(String[] cmd) 指定されたコマンドを実行するための新しいプロセスを生成する
メソッドの概要	
public void	setErrorHandler(CommandErrorHandler handler) エラーイベントハンドラーを登録する
public void	and() バックグラウンドでプロセスを開始する

表 B.4: AOSted で使用できる API とその仕様（基本的機能 2 ）

インタフェース AostedErrorHandler
既知の実装クラスの一覧
LoadErrorHandler, StartAllErrorHandler, StopAllErrorHandler, CollectErrorHandler, CommandErrorHandler

クラス LoadErrorHandler	
コンストラクタの概要	
public	LoadErrorHandler() Load を実行する時のエラーイベントハンドラを生成する
メソッドの概要	
public void	unknownHost(FailEvent e) Load を実行した時、 指定したホストから応答がなかった時に実行される

クラス StartAllErrorHandler	
コンストラクタの概要	
public	StartAllErrorHandler() StartAll を実行する時のエラーイベントハンドラを生成する

クラス StopAllErrorHandler	
コンストラクタの概要	
public	StopAllErrorHandler() StopAll を実行する時のエラーイベントハンドラを生成する

クラス CollectErrorHandler	
コンストラクタの概要	
public	CollectErrorHandler() CollectResult を実行する時のエラーイベントハンドラを生成する

クラス CommandErrorHandler	
コンストラクタの概要	
public	CommandErrorHandler() Command を実行する時のエラーイベントハンドラを生成する

表 B.5: AOSted で使用できる API とその仕様（エラーイベントハンドラクラス）

クラス FailEvent	
フィールドの概要	
public String	hostName 例外が生じたホスト名
public java.util.Date	date 例外が生じた時間
public Throwable	thrownException 生じた例外
public Object	obj 例外が生じた時に実行していたクラスのインスタンス
コンストラクタの概要	
public	FailEvent(String host, java.util.Date date, Throwable e, Object obj) FailEvent オブジェクトを構築する
メソッドの概要	
public String	getHostName() 例外が発生したホスト名を返す
public java.util.Date	getDate() 例外が発生した時間を返す
public Throwable	getThrownException() 発生した例外を返す
public Object	getObject() 例外が発生した時に実行していたクラスのインスタンスを返す
public void	replaceHost(String host) 例外が発生した場合、新しいホストで再試行する

表 B.6: AOSted で使用できる API とその仕様（エラーイベント FailEvent クラス）