継続を基本とする言語 CbC による分散プログラミング

河野真治, 渕田良彦, 宮國 渡 [1]
Shinji KONO, Yoshihiko Fuchita, Wataru MIYAGUNI

継続を基本とする言語 CbC による分散プログラミングについて考察する。CbC は、C に継続を付加し、while 文などのループの制御構造とサブルーチンコールを取り去った言語である。この言語を用いての、分散プログラムの実装とその有効性を述べる。

1 はじめに

分散プログラミングとは、ネットワークなどの比較的疎に結合した環境での並列プログラミングとみなすことが出来る。分散プログラミングには様々な手法がある。ネットワーク API を呼び出す原始的な方法 (Unix socket library など) や、より高度なライブラリ使う (XML SOAP([7])、プログラミング言語外で分散プログラミングを行うこともできる。

また、分散プログラミングに必要な記述を言語仕様に埋め込み、分散プログラミング言語を定義することも行われて来た。(例えば Telescript [9] など)) あるいは、オブジェクトと、そのメソッド呼び出しに着目し、その部分を規格化することによって、分散プログラミングを規定する手法 (CORBA[8] なあど) もある。

様々な手法は、通信のための複雑なセマンティクスを定義することになる。それらは、厳密あるいは曖昧な、日常言語あるいは形式的記述によって定義される。それに従って、プログラムを行い、それに従って、プログラムのデバッグなどを行う必要がある。

本論文では、CbC (Continuation based C) という継続を基本とする言語を用いて、分散プログラミングを行う方法を示す。継続による記述を行うことで、ネットワーク API を呼び出す記述と、その、ネットワーク API そのものを規定する CbC の記述を、ソースコード上で対応する形で定義することが出来る。

これにより、通信のための複雑なセマンティクスの定義そのものが、CbC によって記述され、(決定的な実行を実現するスケジュラーを含めて) 実行可能な形で示される。実際、ネットワーク上で通信する分散プログラムを、対応するシミュレータ記述に、一対一に近い形で変換することが可能である。

この変換は単純であり、変換そのものを分散プログラムのデバッグあるいは検証手法の一つとみなすことが出来る。

2 分散プログラムの例

ここでは、分散プログラムの例として IRC(Internet Relay Chat [3, 4]) のシステムを考える。IRC は一つ以上の IRC サーバでスパニングツリーネットワークを構成し、各サーバに接続した IRC クライアントが join したチャンネル上でリアルタイムにテキストデータを交換するシステムである。

IRC には、サーバ間通信、サーバクライアント間通信、クライアント間通信が存在する。IRC プロトコルには、二つのクライアントが直接にやりとりするのではなく、クライアント間通信は一つ以上のサーバのサーバ間通信によってリレーされる (図 1)。全 IRC サーバは、上位下位サーバの接続情報と、クライアントの情報を持っていなくてはならない。そのため、情報に追加、変更などの処理が行われたら、その都度全てのサーバへ変更の知らせを接続されたサーバを通してマルチキャストする。このマルチキャストは、同時平行的かつ非同期に行われる。このように、分散プログラムでは複数の情報流が非同期的に通信することになる。

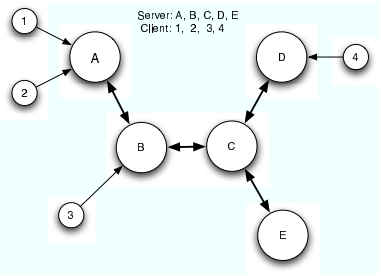

図 1: IRC ネットワーク

[1]琉球大学工学部情報工学科
gongo@cr.ie.u-ryukyu.ac.jp,kono@ie.u-ryukyu.ac.jp
Infomation Engineering, University of the Ryukyus

また、分散プログラムでは単純な通信とは異なり、通信の優先順位や木構造での輻輳制御を行うこともあるため、OSI のトランスポート層に対しての操作が必要となる場合もある。例えば、IRC サーバでは複数のネットワーク接続を監視する必要があり、OSI のネットワーク層とは異なる自分自身のルーティングの管理が必要である。

非同期的な通信は、Unix では通常は、select システムコールを用いて、

```
while(select(fds_max,fds,...,TIMEOUT)>= 0){
  for(int i=0;i<fds_max;i++){
    if(FD_ISSET(i,fds)){
        ......
    }
  }
}
```

のような形のメインループで処理が行われる。

IRC サーバは状態を持つために、同じ通信要求を受け取っても同じ処理をするとは限らない。例えば、クライアントがチャンネルに join する前と後では IRC サーバに送られたテキストの処理は異なる。

このように、分散プログラムは、以下のような複数の要素が全て関わっている。これが分散プログラムを複雑にしている原因である。

1. 通信するプログラム
2. 通信するプログラムの状態遷移
3. 通信モデル
4. 通信プロトコル
5. OSI ネットワーク階層
6. デバイス・ドライバ
7. ネットワーク機器

3 分散プログラムの要素

我々は分散プログラムには三つの要素が必須であると考えている。

- Protocol Engine
- Local Access
- Configuration

プログラム内でこれらの要素を分離することができれば、プログラム開発時、およびテスト時に、それぞれの要素を集中してサポートすることができる。

Protocol Engine は、ネットワーク通信をインターネット上で中継しながら処理を行う部分である。これは、IRC プロトコルそのものと、それを実現する IRC サーバと 相当する。

Local Access は分散アプリケーションを直接に使うユーザ、あるいは、クライアントに対応する部分である。IRC では、IRC クライアントが、これに相当する。

Configuration は、分散アプリケーション、あるいは分散プログラムの地理的、論理的配置を決定する部分である。IRC では、IRC サーバの設定ファイルに相当する。

4 Continuation based C

Continuation based C(CbC) は、C からループ制御構造とサブルーチンコールを取り除き、継続を導入した C の下位言語である。継続呼び出しは引数付き goto 文で表現される。CbC にはサブルーチンコールが存在しないので、通常の継続と異なり継続に環境を含める必要が無い、これは通常の継続よりも小さいので軽量継続ということもできる。

以下の例は、CbC による階乗の計算である。

```
code fact(int n, int result,
          code (*print)()) {
    if (n > 0) {
        result *= n;
        n--;
        goto fact(n,result,print);
    } else {
        goto (*print)(result);
    }
}
```

code は code segment と呼ばれる。これは return 文がないのでサブルーチンではない。引数は interface と呼ばれる。interface は構造体で表現するのが便利であり、struct の代わりに interface というキーワードを用いることもできる。

C 言語に対して、code segment と引数付き goto 文を付加する言語を定義するためには、C のサブルーチンへ戻るための環境付き継続を導入する必要が有る。この言語を CwC(C with Continuation) と呼ぶ。CwC は C の上位言語である。CbC は CwC の仕様の一部に制限されたものとみなすができる。

5 CbC による分散プログラミング

CbC は状態遷移単位での処理の記述が可能である。よって、通信するプログラムの全ての状態を code segment で記述していく。状態遷移は各 code segment からの goto によって表現できる (図 2、3)。

ここでは、通信部分を CbC で、実際に Unix 上で通信を行う記述と、CbC によって Simulate する記述の二種類の記述を行う。

```
code check_command(message *recv) {
  ....
  goto exec();
}

code exec() {
  ...
  if (write_msg) {
    goto send_message(write_msg);
  } else {
    goto reply();
  }
}

code send_message(message *msg) {
  write(msg);
  goto reply();
}
```

図 2: CbC による状態遷移の記述

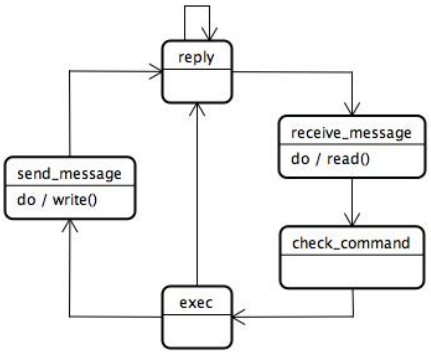

図 3: IRC サーバの状態遷移

6 通信プロトコルの記述

6.1 CbC による最も単純な通信プロトコルの記述

CbC で通信プロトコルを記述するには、通信パケットを interface として記述すると理解しやすい記述となる。

```
interface packet msg;
goto destination(msg);
```

という形でパケットの送信を CbC で表現する事ができる。この方法では、送信先は固定である。

受信側は

```
code destination(interface packet msg) {
  ....
}
```

という code segment となる。これらは、パケットが送信側から受信側に送られるもっとも簡単な表現である (図 4)。

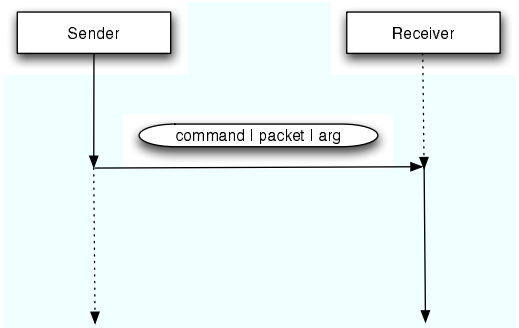

図 4: goto 文で表したパケット

送信先が動的に決まる場合は、msg に送信先を含ませてやれば良い。

6.2 通信ライブラリを含むプロトコルの記述

実際の通信ライブラリを含めて記述する場合は、goto 文が直接通信を表すことはできないので、send

などの通信ライブラリを表す code segment へ goto することになる。

```
int destination;
interface packet msg;
code next();
goto send(destination, msg, next);
```

next は、送信後あるいは送信と同時に実行を行う継続である。

OS の提供する API を使用する場合は、CbC でなく CwC を用い、OS の API を直接サブルーチンコールする。

```
code send(int dest, interface packet msg,
          code (*next)())
{
  write(dest, msg, sizeof(msg));
  goto next;
}
```

受信側は、送信側よりも少し複雑になる。リアルタイムゲームのような場合は、他の処理を行う必要が出てくるため、データ受信による待ち状態に入ることは出来ない。このような場合は、メッセージを待つスレッドを用意する、などの方法があるが、ここでは、select() によるポーリングを使った実装を行う。

7 CbC による分散プログラムのシミュレーション

シミュレーションは、Distributed なプログラムの通信部分や状態遷移部分にスケジューラをはさむことによって実装できる。例えば、IRC サーバや IRC クライアントを一つのタスクとして、メッセージの送信、受信時、あるいは、メッセージによるタスクの状態遷移が起こったとき、スケジューラに制御を渡し、次のタスクを実行する。(図 5)。

今回は、先にネットワーク API を使用した実装を行い、それから、それにそった動作を行うシミュレータ記述を作成した。

実際の分散プログラム開発では、先にシミュレータを使って、分散プログラムをテストしてから、実際の通信 API を組み込む方法が有効であろうと考えられる。

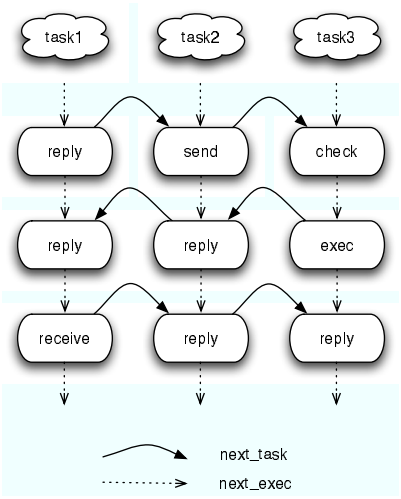

図 5: スケジューラー

8 CbC による IRC のシミュレータ記述

IRC サーバは、他の IRC サーバや IRC クライアントから送られてきた、IRC メッセージに含まれるコマンドに沿った処理を実行する。

以下はシミュレーションの記述である。

```
struct task {
  struct task *next;
  struct message *interface;
  struct irc_server *model;
  code (*exec)();
};

code reply(struct irc_server* s,
           struct task* current_t)
{
  struct message* m;

  if (m = get_message(s->id)) {
    current_t->interface = m;
    current_t->exec = check_command;
  } else {
    current_t->exec = reply;
  }

  goto scheduler(current_t);
}

code check_command(struct irc_server* s,
```

```
                    struct message* m,
                    struct task* current_t)
{
  switch (m->command) {
    case SERVER:
      current_t->exec = add_server;
      break;
    case NICK:
      current_t->exec = set_nickname;
      break;
    case JOIN:
      current_t->exec = channel_join;
      break;
    default:
      break;
  }
  goto scheduler(current_t);
}
```

この記述では、自分のサーバに IRC メッセージが送られてきたかどうかを get_message() で確認し、そのメッセージを元に次に遷移する状態を決めている。

このプログラムでは、もちろん、複数のタスクが同時に動作することをシミュレーションする必要がある。これは、スケジューラを用いることで実現できる。scheduler() は、カレントタスクを渡し、次に実行するタスクを呼び出す。

C などの軽量継続を持たない言語では、スケジューラを直接記述することは出来ない。例えば、ABCL/1 [5] の用に、プロシジャーを分割し、その分割したプロシジャーを順に呼び出すような手法が必要となる。CbC では、この分割が code segement 単位で前もって実現されているので、一対一の記述が可能になっている。

```
code scheduler(struct task* current_t)
{
  struct task* next;

  next = get_next_task(current_t);
  goto next->exec(next->model,
                  next->interface, next);
}
```

task->model は、そのタスクが操作するオブジェクトで、task->interface は、オブジェクトに対する引数である。

Distributed なプログラムでは、get_message() の代わりに select() から、ソケットの状態を確認してメッセージの有無を調べる方法が一般的である。また、状態遷移では、カレントタスクに次の状態を設定するのではなく、直接 goto すればよい。

```
code reply(struct irc_server* s)
{
  struct message* m;

  if (m = get_message(s->id)) {
    goto check_command(s, m);
  } else {
    goto reply(s);
  }
}
```

9 二つの記述の比較と使用法

今回は、実際に通信するプログラムを先に作成し、それから、シミュレータ記述を作成した。これらのプログラムは、一対一の記述になっているが、もちろん、別な構造を持っている。

実際に通信を行うプログラムは、当然、複数独立に動作するプログラムであり、複数の main() を持つ。Unix API はサブルーチンとして呼び出され、その動作は、Unix API の定義によって決まっている。

シミュレータ記述では、main() は単一であり、通信を行う複数独立のプログラムに相当する複数のスレッドを code segement として含んでいる。さらに、Unix API をシミュレートする部分と、スケジューラを持っている。

もとの通信するプログラムからシミュレータ記述への変換は、[1] では、平坦化と呼んでいる。これは、[6] の relativization と呼ばれる、複数のオートマトンを一つにまとめることにも対応している。

このシミュレータを作成する過程で、IRC サーバの初期化が終る前に、通信がある場合に、バグがあることが判明した。

このシミュレータの動作を、例えば、モデル検証ツールや、タブロー展開ツールなどで、網羅的に調べることが出来れば、分散プログラムのテストや検証を実現することが出来る。

10 まとめ

本稿では、継続を基本とする言語 CbC を用いた、分散プログラムの記述法について述べた。

通信 API が組み込まれているプログラムと、シミュレーションの記述の互換性は、それほど大きくなく容易である。

この変換を用いて、分散プログラムのテストや検証を実現することが出来ると考えている。

参考文献

[1] 楊挺, 河野真治. “ 継続を基本とする言語 CbC による分散計算 ”. 沖縄情報通信ワークショップ, Nov, 2002.

[2] 安村恭一, 河野真治. “ 動的ルーティングによりタプル配信を行なう分散タプルスペース Federated Lind ”. 日本ソフトウェア科学会第 22 回大会論文集, Sep, 2005.

[3] http://www.faqs.org/rfcs/rfc2810.html. Internet Relay Chat: Architecture

[4] http://www.faqs.org/rfcs/rfc2813.html. Internet Relay Chat: Server Protocol

[5] A. Yonezawa and E. Shibayama and T. Takada and Y. Honda, Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1, Object-Oriented Concurrent Programming, 1987, MIT Press

[6] P. Wolper, Synthesis of communicating processes from temporal logic specifications Dept. of Computer Science, Stanford University, STAN-CS-82-925, 1982

[7] Nilo Mitra (Ericsson), SOAP Version 1.2 Part 0: Primer, W3C Recommendation 24-June-2003.

[8] CORBA/e Draft Adopted Specification

[9] Cronder Concepcion , Paul Staniforth , Byte Guide to Telescript, 1995