継続を基本とする言語 CbC による分散計算
Using Continuation based C for Distributed Computation

楊 挺† Yang Ting　　河野 真治‡ Shinji KONO

1. まえがき

継続を基本とする言語 CbC による分散計算について考察する。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とりさった言語である。この言語を用いて、分散計算の実装および仕様記述をおこなった。他の仕様記述言語との比較 (Lotos, System C) などとの簡単な比較もおこなう。

2. 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つき 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ない。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。また、C を変換して CbC で表現することも可能である。[1] この場合は、環境は継続の引数として明示的に表現する必要がある。以下に、CbC の例題として `g(i)` を呼び出す C の関数 fact を CbC に変換した例を示す。STACK sp が明示された環境を格納する。

```
typedef char * STACK;
struct f_save{
    code(*ret)(STACK,int);
    int i,k;
};
code fact(STACK sp,int i){
    int k;
    struct f_save *c
    k=3;
    sp -= sizeof(struct f_save);
    c = sp;
    c->i = i;
    c->k = k;
    if( k>i )
        goto g(sp,i);
    else goto (*c->ret)(sp,i);
}
```

code は code segement 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。STACK などの code segment の引数は、code segment の interface と呼ばれる。interface は構造体で表現するのが便利であり、struct の変わりに interface というキーワードを用いることもできる。

継続は基本的には goto 文である。これを用いることにより複数のオブジェクトへのメッセージ送信をおなじ道を通ることなく伝えることができる。RPC などでは呼び出しの答えは必ず呼出側に返る必要がある。(fig.1) このように、継続はサブルーチンコールよりも通信と相性が良い。分散オブジェクトでのメッセージ送信も実体は RPC である場合が多い。

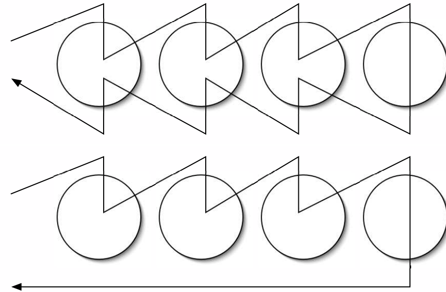

図 1: RPC と goto

本稿では、この CbC を用いて分散計算 (ネットワークプログラミング) を行う場合の枠組と、その利点および欠点について考察する。

3. 分散アプリケーションの要求と要素

ここでは分散アプリケーションの例として IRC (Internet Relay Chat) のシステムを考える。IRC(Internet Relay Chat) は、複数のサーバで木構造を構成し、サーバに接続した IRC クライアントにより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。チャットは複数のチャンネルから構成され、クライアントは join するチャンネル上でテキストを交換する。

IRC には、サーバ間の通信とクライアント間の通信の二種類が存在する。あるチャンネルに入力されたテキスト情報は木構造を必要なところまで遡り、そのチャンネルを購読するクライアントに配付される。ユーザ名やユーザが join したチャンネルの変更の情報は全てのサーバに転送される必要がある。(図 2) このように分散アプリケーションでは複数の情報流が非同期的に通信することが多い。また、その情報流の優先順位はアプリケーションにより動的に決定される。

また、通信は広域ネットワーク上で行われるため OSI の 7 階層により処理される。しかし、単純な通信と異なり通信の優先順位や木構造での輻輳制御を行うためには、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。これは、例えば Unix などではソケットに対するオプションの設定などを通して行われる。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。

Unix では通常は、

```
while(select(max_fds,fds,...,TIMEOUT)>=0) {
```

†琉球大学理工学研究科
‡琉球大学情報工学科,
科学技術振興事業団さきがけ研究 21(機能と構成) 領域

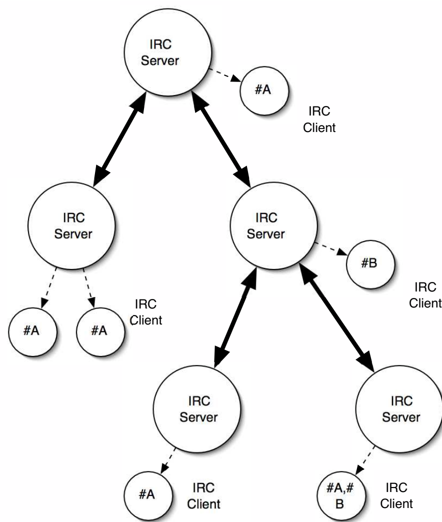

図 2: IRC

```
        for(int i=0;i<MAX_FDS;i++) {
            if (FD_SET(fds[i])) {
                ....
            }
        }
    }
```

のような形のメインループで処理が行われる。サーバは状態を持つためにおなじパケットを受け取ってもおなじ処理をするとは限らない。例えば、チャンネルに join する前と後では IRC に送られたテキストの処理は異なる。

通信自体は、read/write のような比較的単純な API を使う。これらの API を通過した後は、Unix カーネル内部のトランスポート層の実装を経由してネットワーク機器にアクセスするデバイス・ドライバに引き渡される。逆に、ネットワークから到着したパケットは、デバイスドライバ経由でアプリケーションのメインループにイベントとして報告される。

このような Unix の API レベルでのネットワークプログラムは繁雑である。これを簡素化するために様々な通信モデルが提案されている。例えば、RPC(Remote Procedure Call), Message Passing, ORB (分散オブジェクト管理), Tuple Communication , MPI, Grid などである。これらは通常は前述のメインループを持っていることが多い。しかし、IRC や電子メールなどのアプリケーションは効率を優先するために直接 Unix API を用いていることが多い。

個々のサーバでのプログラムは、IRC 全体のプロトコルを定義する。このプロトコルは、通常は RFC などの文書で別途記述される。プロトコルで特に重要なのは、実際に通信されるパケットの形式である。また、これらのプロトコルが途中で止まるなどの不都合が無いようなプロトコルの整合性が要求される。さらに、あらゆる分散アプリケーションは、現状ではなんらかの攻撃の対象になると考えられる。プロトコルとその実装は、可能な攻撃に対して対応できるものである頑強性が要求されている。

この例で見たように分散アプリケーションは、以下のような複数の要素が全て係わっている。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

4. CbC による分散計算のサポート

CbC は C の下位言語であり、分散計算に対するサポートは従来のプログラミング言語によるサポートとは異なる。 CbC の記述はアセンブラに近くアプリケーション・プログラムやサーバを直接記述するのには向いていない。一方で、実装言語としての使い方だけでなく、状態遷移記述を用いたハードウェア記述やプロトコル記述が可能である。したがって以下のような用途が考えられる。

```
実装言語のコンパイラターゲット
通信プロトコルの記述
デバイス・ドライバの実装
OS 内部 (トランスポート層など) の記述
ネットワーク機器の記述とシミュレーション
通信モデル自体の記述
```

実装言語としては、現状では C,C++や Java が用いられることが多いと思われる。CbC は、これらの言語のコンパイラターゲットとして用いることが可能であり、他のネットワーク階層の CbC 記述と整合させることができる。

5. CbC による分散計算の実装

CbC を実装言語のターゲットとして用いる場合は、通信部分を CbC で実装することになる。この部分を実装する方法には以下の二種類が考えられる。

```
OS の API を呼び出す形での実装
OS 内部を CbC で記述し、その記述と接続する方法
```

Unix などの汎用 OS では前者が良く、Real-time モニタのような場合は後者による実装が効果的である。

後者の場合は、CbC は (アセンブラ抜きで) 実行コンテキストそものを取り扱うことができるので、read/write あるいは select 時の待ち状況を自分自身で表現することができる。例えば、特定のメッセージを待つ場合には以下のような方法で他のスレッドに切替えることができる。

```
code wait_thread_a(interface a self,schedulder s) {
    code wake_thread_a(interface a,scheduler s);
    extern int volatile message *msg;
    if (msg) {
```

```
        a->message = msg;
        goto wake_thread_a(interface a,scheduler s);
    } else {
        a->continuation = wake_thread_a;
        (interface a)(s->current) = a;
        goto s->next_thread(s);
    }
}
```

CbC は、サブルーチンコール用のスタックフレームを持たないため、このような記述が可能となっている。C などではアセンブラを使用するか setjmp などのシステム依存の API を使用する必要がある。Thread をサポートした言語 (Java あるいは Pthread) では、対応する複雑な API を使用する必要がある。

通信部分が実際のデバイスを対象としている場合、例えば、デバイス記述そのものである場合は、CbC の持つ状態が限られているので、割込みやイベントなどを処理する code segment を状態の分だけ記述してやれば良い。

6. CbC と C の混在

CbC の上位言語として C の機能を含む CwC (C with Continuation) と言う言語も用意されている。CbC と C を混在させるためには、CwC を使用するだけでよい。

しかし、ハードウェア記述などと接続する場合は、C を CbC にコンパイルして、CbC だけで記述する方が直接的である。この場合は、最初の例のように明示的なスタックが存在するので、それを適切に処理する必要がある。マルチスレッドなプログラムでは、そのスタックを複数、スケジューラーが管理する必要がある。これは、基本的にはユーザレベルスレッドの実装と同様である。

7. 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場合は、通信のパケットを interface として記述する。パケットの中身だけでなく、goto 文の行き先も指定できる。この goto 文は、分散計算の表現であり実装でなくても良い。つまり、

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

という形でパケットの送信を CbC で表現する事が出来る。しかし、この方法では、通信先は固定である。受信側は、

```
code destination(interace packet contents) {
  ....
}
```

という code segment となる。これは、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 (refpacket)

8. 通信ライブラリの入るプロトコルの記述

送信先が動的に決まる場合は、contents に送信先を含めてやれば良い。destination code segement で受信先をシミュレーションすることが可能である。

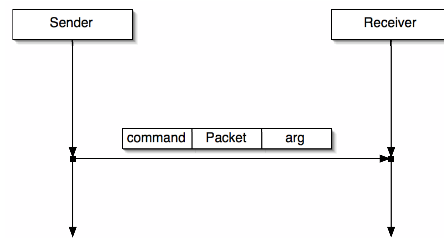

図 3: goto 文で表したパケット

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 segement へ goto することになる。

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

ここで、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。

受信側では状況は、それほど単純にはならない。受信側は単純に送信を待つことも出来るが、リアルタイム・ゲームのような場合は受信側が他の仕事をしている場合があるためである。実際、OS のセッション層では、TCP/IP のポートを使ったアプリケーションへの振り分けが行われている。アプリケーション内部でも、受信した内容に合わせて処理を行う必要がある。また、その処理は、現在実行中のスレッドの状態変化を生じさせるものであることが多い。

このような場合は、前述のようにメインルーチンを持ち、登録されたイベントに対してコールバックを行うことが多くの上位言語で行われている。また、スレッドを持つ言語の場合は、メッセージを待つスレッドを用意することが一般的である。どちらの方法でも CbC を用いて記述することができるが、基本的にはポーリングを使用した形での記述となる。

```
code message_check(interface a) {
    extern volatile message *msg;
    if (msg) {
        goto msg->hadler[msg->type](msg,a);
    } else {
        goto (a->next)(a);
    }
}
```

前節のようにスケジューラをはさむことでより柔軟なイベント処理を行うことができる。

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方がやさしい。あるいは、OS の API を記述し、シミュレーションする形で分散アプリケーションを記述することも可能である。ただし、API を使用する場合とシミュレーションする場合の記述の互換性は、それほど大

きくないので、プログラム変換などの手法が必要になると思われる。

9. 通信プロトコルの平坦化

前節のように通信ライブラリを間にはさむと、通信プロトコルそのものは判読しにくくなる。通信主体と状況を固定すれば、その特定のプロトコルの状況は、より簡易な形で記述することが可能である。つまり、通信ライブラリを含んだ分散計算の記述を、より単純な位置依存性のないプロトコルの記述に変換することが可能である。これを通信プロトコルの平坦化という。

平坦化された通信プロトコルは、より単純な状態遷移に落ちるので、これに対して、モデル検証 [6] や、時相論理による検証 [7] を使った検証をおこなうことが可能である。(図 (refflatten)

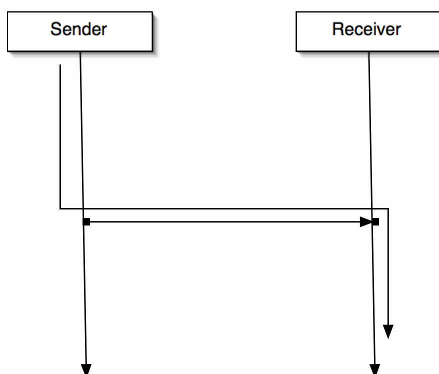

図 4: 通信プロトコルの平坦化

10. 記述の例題 (IRC)

前述の IRC では、クライアントはテキストをサーバに送り、サーバはチャンネルリストを参照しながら自分の知るサーバにテキストを送信する。IRC server は自分の上位のサーバのチャンネルリストを irc.list に持っているとする。以下は平坦化された記述である。

```
interface IRC {
    char *channel;
    server *parent;
    char *text;
}
code IRC_client(interface IRC irc) {
    goto IRC_server_text(irc);
}
code IRC_server(interface IRC irc) {
    if (member(irc.channel,irc.parent)) {
        goto IRC_send_sever(irc);
    } else {
        goto IRC_client(irc);
    }
}
code IRC_send_server(interface IRC irc) {
    goto IRC_server(irc.parent);
}
```

この記述では、`IRC_client` で `irc.text` がサーバに送られ、サーバ側で上位に送るかどうかを `member` 関数で調べている。ここでは `member` 関数は C の関数である。この部分を CbC で記述するには一段継続をはさめばよい。

```
code IRC_server(interface IRC irc) {
    goto  member(irc,IRC_server_1);
}
code IRC_server_1(interface IRC irc) {
    ...
}
```

のような形になる。

11. 他の記述/実装言語との比較

プロトコル記述言語としては、LOTOS がしられている。LOTOS は複雑な公理系を持つプロセス代数の記述である。LOTOS を実装言語として用いることは出来ない。CbC は、単純な goto 文のセマンティクスを持ち、構文もほぼ C なので実装者が理解しやすい。

SPEC C[8] は、システム記述用の C 言語の拡張である。この言語も並列処理記述用の極めて複雑なセマンティクスを持っている。CbC では複雑な並列処理を記述する場合は、スケジューラを含めて明示的な記述を行う。

CbC は、コンパイラの実装と C への変換の実装を持っている。また、既存の C から CbC へ変換することも出来るので、いままでのソフトウェアとの親和性が高い。また、記述言語として用いた場合にも直接実行できるという利点がある。

12. まとめ

本稿では、継続を基本とする C に近い言語 CbC を用いた分散計算の記述法について述べた。CbC は goto 文のみを持つ単純な実行セマンティクスを持ち、コンパイラターゲットや、状態遷移記述、仕様記述言語として用いることが出来る。

また、goto 文を通信と解釈することにより、分散計算の記述と実行を行うことが可能である。

参考文献

[1] 河野 真治、楊挺:SWoPP 2001,Okinawa,July,2001 C 言語の Continuation based C への変換

[2] 河野真治 日本ソフトウェア科学会第 19 回大会論文集継続を基本とした言語 CbC の gcc 上の実装

[3] 佐渡山 陽 平成 13 年卒業論文 PS2 の DSP プログラムを記述する言語に関する研究

[4] http://www.faqs.org/rfcs/rfc1459.html Internet Relay Chat Protocol

[5] http://www.faqs.org/rfcs/rfc1831.html Remote Procedure Call Protocol

[6] J.R. Burch, E.M. Clarke, K.L. McMillan and D.L Dill, Sequential Circuit Verificaton Using Symbolic Model Cheching, Proc. 27th ACM/IEEE Design Automation Conf. Jun, 1991

[7] Shinji Kono, A Combination of Clasual and Non Clausal Temporal Logic Program IJCAI-93 Workshop on Executable Modal and Temporal Logics,Aug, 1993

[8] Daniel D. Gajski , Jianwen Zhu , Rainer Dmer , Andreas Gerstlauer , Shuqing Zhao SPEC C:SPECIFICATION LANGUAGE AND METHODOLOGY KLUWER ACADEMIC PUBLISHER", 1999