平成 25 年度卒業論文

mruby を用いた Linux ロードバランサインタフェースの実装

電気通信大学 情報理工学部
情報・通信工学科
コンピュータサイエンスコース

学籍番号 : 1011209
氏名 : 吉川 竜太
指導教員 : 中野 圭介 准教授
提出日 : 平成 26 年 1 月 31 日

要旨

サーバ台数の増加に伴い，ロードバランサと呼ばれるサーバの負荷分散システムと，プログラムを用いたサーバ運用の自動化が普及している．Linux におけるロードバランサとして，IP Virtual Server (以下 IPVS) があるが，IPVS の操作に用いられる既存のインタフェースは，設定の記述性や自由度に制限があるため，ロードバランサの操作をプログラムを用いて自動化するのが困難である．この問題に対処するため，本研究では，軽量 Ruby である mruby から IPVS を操作するためのインタフェースを設計し，実装した．Ruby は記述力が高く，サーバ運用の自動化をする際によく用いられるため，サーバエンジニアの間で普及が進んでいる．Ruby と同程度の記述力を持つ mruby を用いることで，高い記述力で IPVS を操作できる．また，拡張モジュールを使用することによる独自のヘルスチェック定義が可能である．mruby は軽量であるため, 速度等でも既存のインタフェースに劣ることなく，サーバサイドで長時間動くことが想定されるロードバランサの操作を，自動化しやすくすることを実現した．

目次

1 はじめに

1.1 背景

サーバ台数の増加に伴い，**ロードバランサ**と呼ばれるサーバの負荷分散システムと，プログラムを用いたサーバ運用の自動化が普及している．

Linux における代表的なロードバランサとして，IP Virtual Server [1] (以下 IPVS) がある．IPVS は，クライアントからのリクエストの振り分けの設定を行うために，C 言語の API を呼び出す必要がある．そのため，一般的には Keepalived [2] などの設定用のソフトウェアを用いることが多い．Keepalived は，設定ファイルに独自の構文を採用しているが，次のような問題点が挙げられる．

- 構文チェックを行わない
  設定ファイルに対する構文チェックを行わないため，設定ファイルに誤った記述がなされた場合，意図しない振り分け設定でロードバランサが動作する可能性がある．
- 記述力が低い
  設定ファイルに制御構文を使えないため，様々な条件による処理の振り分けを行うのが困難である．
- 拡張性が低い
  ライブラリの追加は行えず，独自の構文を定義することも出来ないため，設定ファイルを拡張することが困難である．

前述のような問題点があるため，Keepalived を用いたロードバランサの操作では，ロードバランサの操作を間違いなく細かく記述するのが難しい．また，条件分岐を使った動的な振り分け先のサーバ追加や，繰り返し構文を使った大規模なサーバ群の処理などが行えないため，ロードバランサの操作を自動化するのが困難となっている．

1.2 目的と方針

本研究では，軽量スクリプト言語である mruby [3] から IPVS を操作可能なインタフェースを設計し，これを mruby のサードパーティライブラリである mrbgem として実装する．mrbgem は，C 言語で書かれた拡張ライブラリを mruby のビルド時に組み込むことを可能にするものである．これは，mruby から 既存の C 言語アプリケーションを操作する際によく用いられる手法である．

mruby は 一般に用いられている Ruby の軽量版であり，Ruby とほぼ同程度の記述性や生産性を持っている．また，軽量版であるため，ロードバランサのようなソフトリアルタイムなシステムに適していると考えられる．

本機構により，既存の IPVS インタフェースの問題点を解決し，ロードバランサの操作を自動化しやすくすることを目的とする．

1.3 本論文の構成

本論文は，次のような構成からなる．2 章では，ロードバランサの概要ならびに Linux のロードバランサである IPVS について説明する．3 章では，既存の IPVS インタフェースである Keepalived ならびに ipvsadm に関して説明する．4 章では本システムの設計について，5 章では実装について述べる．6 章で本システムの評価を行い，7 章で関連研究，8 章で本論文のまとめと今後の課題について言及する．

2　ロードバランサ

本章では，ロードバランサの役割ならびに，ロードバランサが一般的に備えている機能の概要について記す．

2.1　ロードバランサの役割

ロードバランサは，サーバの負荷を分散させ，サービスの拡張性 (scalability) と可用性 (availability) を高めるために用いられるシステムである．図 1 に，ロードバランサの概要図を示す．ロードバランサの主な機能として，リクエストの振り分け，セッション維持，**ヘルスチェック**が挙げられる．

ロードバランサには，通信機能の階層の国際規格である，OSI 参照モデルにおけるレイヤ 4 ベースの L4 ロードバランサと，レイヤ 7 ベースの L7 ロードバランサが存在する．L4 ロードバランサではパケットのトランスポート層までの情報を用いて負荷分散を行い，L7 ロードバランサではアプリケーション層までの情報を用いる．そのため，L4 ロードバランサは L7 ロードバランサに比べ高速に動作するが，L7 ロードバランサでは L4 ロードバランサに比べより高度な負荷分散を行うことが可能である．

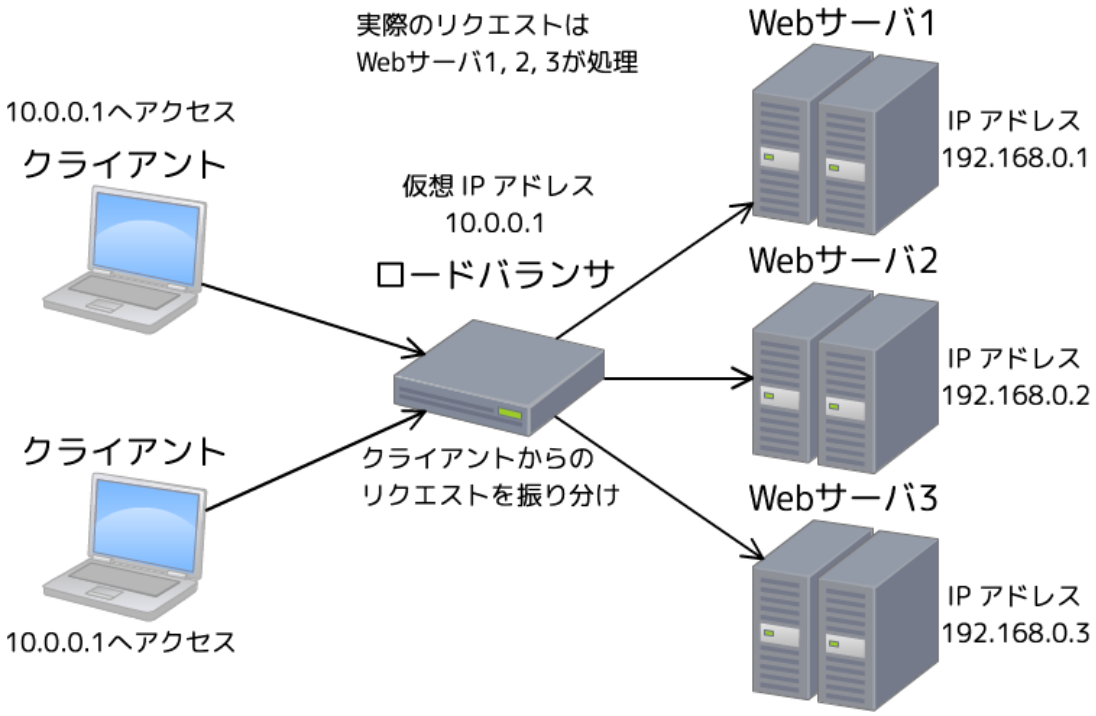

図 1　ロードバランサの役割

2.1.1 リクエストの振り分け

一般的なロードバランサは，サービスに用いる特定の IP アドレス (**仮想 IP アドレス**) を定める．仮想 IP アドレスにてクライアントからリクエストを受け取り，同等の機能を持つ複数のサーバへ振り分けを行う．この時，振り分けられるサーバ群を**実サーバ (Real Server)** と呼ぶ.

リクエストの振り分けにより，特定のサーバにリクエストが集中し負荷が上昇することを防ぐことが可能であるため，サービスの可用性が向上する．サービスの拡張が必要となった際には，同等の機能を持つサーバを構築してサーバの数を増やし，ロードバランサの振り分け先に追加する作業を行うことで，サービスの拡張性を向上させることができる.

ロードバランサがクライアントからのパケットを実サーバに転送する方法の例として，次のようなものが挙げられる.

- NAT (Network Address Translation)
  パケットの送信元・送信先アドレスおよびポートを書き換えて転送する方法.
- ダイレクト・ルーティング
  パケットを操作することなく直接振り分け先のサーバへ転送する方法.
- IP-IP カプセル化 (トンネリング)
  送信されてきたパケットのイーサネットフレームを，ロードバランサが IP パケットにカプセル化し，振り分け先のサーバへ転送する方法.

NAT を用いた場合，ロードバランサがパケットを書き換えるため，実サーバにて設定を行う必要がなく，比較的容易に負荷分散を行うことが出来る．しかし，実サーバからの戻りパケットもロードバランサを経由する必要があるため，ロードバランサのネットワーク帯域がボトルネックとなることが多い．ダイレクト・ルーティングを用いた場合，IP パケットは操作されないため，実サーバがロードバランサの IP アドレス宛のパケットを受け取れるように設定する必要があるが，実サーバからクライアントへの通信は，ロードバランサを経由しないため，高速に動作する．この様子を図 2 に示す．トンネリングを用いた場合，ダイレクト・ルーティングとほぼ同じ動作をするが，ロードバランサが IP パケットにカプセル化することにより，振り分け先のサーバが別のネットワーク上にある場合でもパケットを転送可能である.

ロードバランサへのリクエストを実サーバに振り分ける方法は，ロードバランサによって定められたアルゴリズムによって決定される．例として，次のようなアルゴリズムが挙げられる.

- ラウンドロビン (Round Robin)
  均等にリクエストを振り分ける.
- 重み付きラウンドロビン (Weighted Round Robin)
  予め定められた比率に従って，リクエストを振り分ける.
- 最小接続 (Least-Connection)
  接続数が少ない実サーバへ優先的にリクエストを振り分ける.
- 重み付き最小接続 (Weighted Least-Connection)

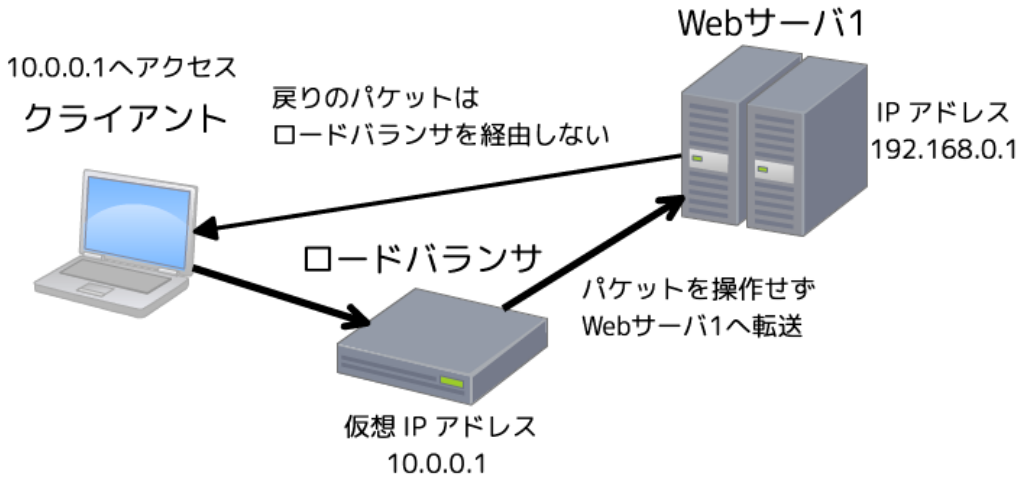

図 2　ダイレクト・ルーティング

予め定められた比率に対し，接続数が少ない実サーバへ優先的にリクエストを振り分ける．

- 所在ベース最小接続 (Locality-Based Least-Connection)
  特定の IP アドレス宛のリクエストを特定の実サーバへ振り分ける．
- レプリケーション付き所在ベース最小接続 (Locality-Based Least-Connection with Replication)
  特定の IP アドレス宛のリクエストを特定の実サーバ群の接続数が少ないサーバへ振り分ける．
- 宛先ハッシュ (Destination Hashing)
  宛先 IP アドレスにより割り当てられたハッシュテーブルに従ってリクエストを振り分ける．
- 送信元ハッシュ (Source Hashing)
  送信元 IP アドレスにより割り当てられたハッシュテーブルに従ってリクエストを振り分ける．
- 最小遅延予測 (Shortest Expected Delay)
  重みと接続数を用いて，遅延が最も少ないと予想されるサーバへリクエストを振り分ける．
- キュー無し (Never Queue)
  有効な接続がないサーバへ優先的にリクエストを振り分ける．
- コンテンツスイッチング (Contents Switching)
  L7 ロードバランサにおいて HTTP の GET や POST の内容や，Cookie の内容などのコンテンツ情報に従って，リクエストを振り分ける．

ラウンドロビン，重み付きラウンドロビンや宛先ハッシュなどのように，あらかじめ決定された順序に従いリクエストを振り分けるものを**静的分散方式**と呼ぶ．これは，リクエスト毎の処理速度が一定である場合などに効果的である．最小接続，重み付き最小接続や最小遅延予測などのように，振り分け時の実サーバの状態により，振り分け先が変わるものを**動的分散方式**と呼ぶ．

これは，リクエスト毎の処理速度が異なる場合などに効果的である．

2.1.2 セッション維持機能

ロードバランサを用いた場合，リクエストが複数のサーバに振り分けられるため，クライアントが接続のたびに別のサーバにアクセスする可能性がある．しかし，クライアントからのリクエストが前回のリクエストと異なるサーバに振り分けられると，不都合が生じる場合がある．

例として，セッション ID をもとにユーザを判別するような Web アプリケーションを考える．クライアントからのリクエストが，前回のリクエストと異なる Web サーバに振り分けられた場合，サーバはクライアントを正しく判別することができない．これは，Web サーバ間でセッションなどの情報を共有することで回避することができるが，このような機能をアプリケーションに追加するのは開発者への負担がかかる．

この問題に対処するため，ロードバランサにはセッション維持機能がある．セッション維持機能を利用すると，あるクライアントのリクエストを一定期間，同じサーバに振り分けることが可能である．そのため，Web アプリケーションを開発する際に，複数のサーバ間で情報を共有する機能を Web アプリケーションに追加することなく，リクエストの不整合が発生するのを回避することが可能である．

2.1.3 ヘルスチェック

一般的なロードバランサは，定期的に振り分け先のサーバのサービス状態を確認し，振り分け先のサーバに異常が生じた場合に振り分け先から外す**ヘルスチェック**と呼ばれる機構を備えている．ヘルスチェックの動作を図 3 に示す．

ヘルスチェックは，異常が発生しているサーバを素早く検知し，リクエストの振り分け対象から外すことにより，サービスの可用性を高めるものである．ヘルスチェックが存在しなかった場合，異常が発生したサーバにもリクエストを振り分けることとなる．例えば，振り分け先のサーバが 3 台存在し，そのうち 1 台に異常が発生していた場合，ラウンドロビンアルゴリズムを用いると，3 回に 1 回のリクエストは正常でないサーバに送られ，誤ったレスポンスが返されたり，そもそもレスポンスが返されないといった事態が起こりうる．

ヘルスチェックは，一般的には OSI 参照モデルのレイヤ 3，レイヤ 4，レイヤ 7 の 3 段階で行われる．レイヤ 3 でのヘルスチェックは，IP ネットワークにおいて，実サーバに到達出来るかどうか ping を用いて応答の有無を検査するものである．レイヤ 4 でのヘルスチェックは，実サーバの TCP および UDP ポートにリクエストを送信し，応答の有無を検査するものである．レイヤ 7 でのヘルスチェックは，アプリケーション自体の応答を検査するものであり，HTTP や FTP，SMTP 等のリクエストコマンドを送信し，応答の有無を検査するものである．

2.2 IP Virtual Server

IPVS は，Linux カーネルに実装された L4 ロードバランサである．1998 年に Wensong Zhang によりオープンソースプロジェクトとして制作が開始された．現在では，IPVS は標準の Linux カーネル 2.4 ならびに 2.6 以降に導入されている．IPVS を用いれば，Linux 上で TCP

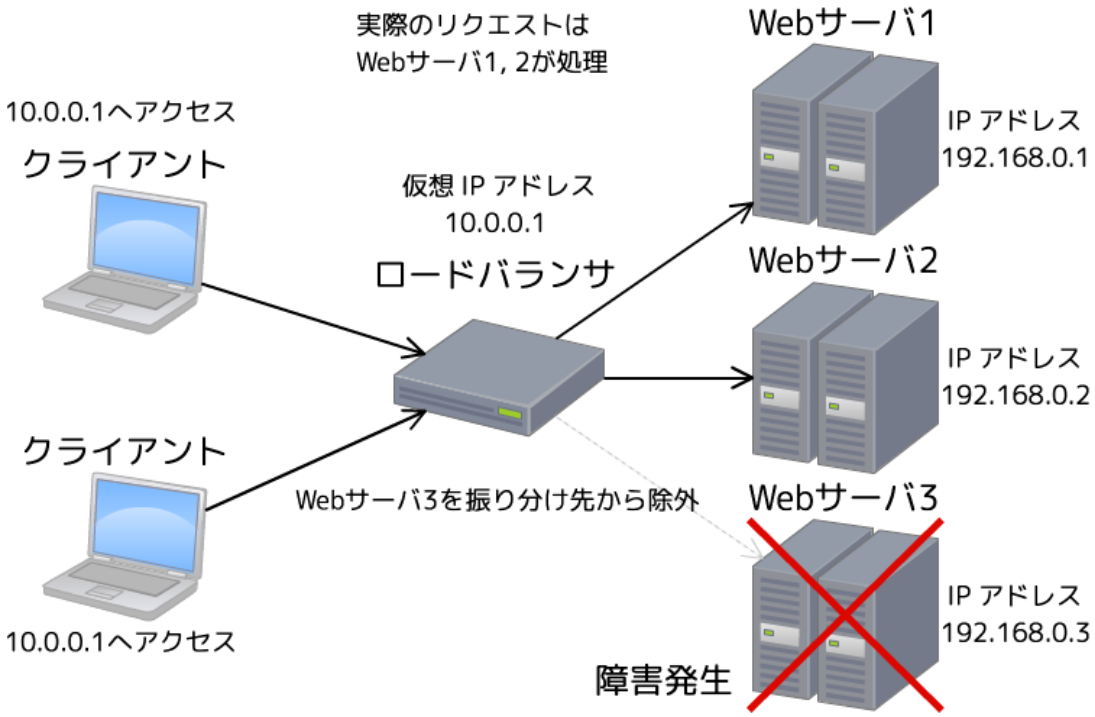

図 3　ヘルスチェックを行うロードバランサ

と UDP のリクエストを複数のサーバに振り分けることが可能である．但し，IPVS にはヘルスチェックの機構が存在しないため，Keepalived などのヘルスチェック機構が付属しているソフトウェアを用いるなどの方法で，異常のあるサーバを振り分け先から外す操作を別途行わなければならない．

IPVS は Linux カーネルに実装されているため，リクエストの振り分け先を追加する操作などは，C 言語を用いてカーネル内の構造体に IP などの情報を書き込む必要がある．そのため，ユーザ空間においては Keepalived や ipvsadm といった，IPVS を操作することが可能なソフトウェアを用いて振り分けの設定を行うことが一般的である．

2.2.1　リクエストの振り分け

IPVS の実サーバへのパケットの転送は，次の方法を使用することが可能である．

- NAT
- ダイレクト・ルーティング
- IP-IP カプセル化

振り分けには，次のアルゴリズムを使用することが可能である．

- ラウンドロビン

- 重み付きラウンドロビン
- 最小接続
- 重み付き最小接続
- 所在ベース最小接続
- レプリケーション付き所在ベース最小接続
- 宛先ハッシュ
- 送信元ハッシュ
- 最小遅延予測
- キュー無し

IPVS では，カーネルモジュールとして振り分けアルゴリズムを追加できる．そのため，上記以外のアルゴリズムを自分で定義して使用することが可能である．

Linux では，netfilter というパケットフィルタリングを行うカーネルモジュールを用いて，特定の条件を満たしたパケットに対して firewall mark と呼ばれる数値型のマークを設定することができる．IPVS では，firewall mark に対して振り分けを追加することができる．例えば，netfilter を用いて 80 番ポートと 443 番ポートを通過するパケットに 1 というマークを付与した場合，IPVS を用いて 1 のマークが付いたパケットを振り分けることが可能である．これにより，異なるポート間でセッション維持機能を利用することが可能であり，別のプロトコルでも同じ実サーバに振り分けられる必要があるようなアプリケーションの場合に有用である．

2.2.2 ipvs syncmaster

ロードバランサを 1 台のみで運用した場合，ロードバランサ自身が単一障害点となるため，障害が発生した場合，サービスが停止することとなる．そのため，IPVS ではロードバランサ自身が冗長化されることを考慮し，IPVS の接続情報を共有するための ipvs syncmaster と呼ばれる機構が存在する．

IPVS は，セッション維持機能のために，次のようなクライアントと実サーバの対応付けをカーネル内に保存する．

- クライアントの IP アドレス・ポート
- サービスの IP アドレス・ポート
- 振り分け先サーバの IP アドレス・ポート

クライアントからリクエストが来た場合，上記の対応付けが保存されているかどうかを確認する．対応付けが保存されているリクエストだった場合は，保存された情報に従い実サーバへ振り分ける．対応付けが保存されていなかった場合は，クライアントと実サーバの新しい対応付けをカーネル内に保存する．

この情報を，動作しているロードバランサから，動作していないロードバランサに定期的に送信する仕組みが ipvs syncmaster である．ipvs syncmaster の動作を図 4 に示す．この仕組みが動作していなかった場合，図 4 の状況において，ロードバランサ 1 に障害が発生し，ロードバランサ 2 へ仮想 IP アドレスを付け替えた際に，クライアント 1 と Web サーバ 1 のセッショ

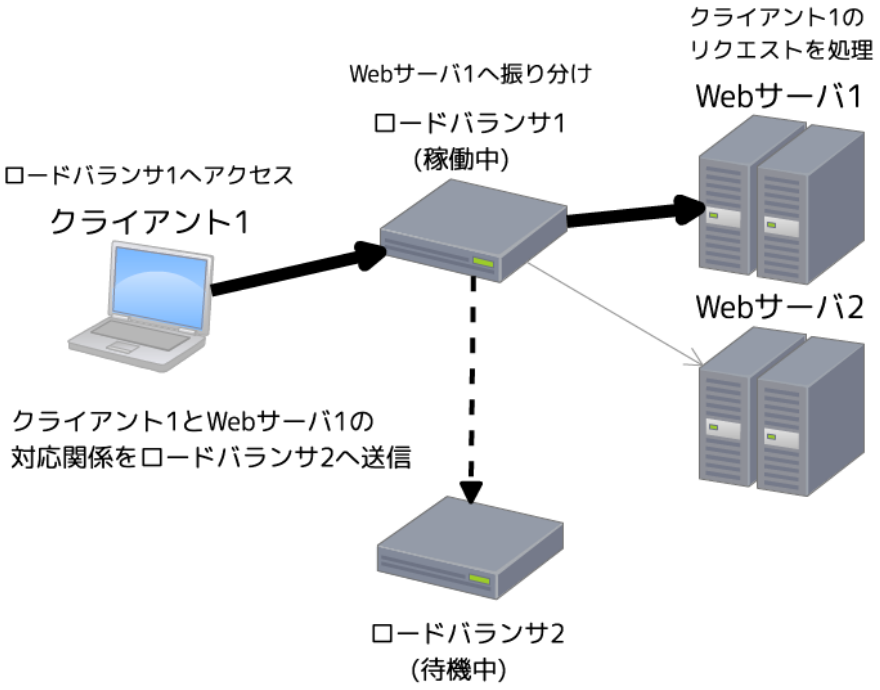

図 4　ipvs syncmaster

ン情報が失われ，次の接続時にはクライアント 1 が Web サーバ 2 に振り分けられ，不正なレスポンスをクライアントに送信してしまう危険性がある．

2.2.3　利用例

IPVS は Linux カーネルの中に組み込まれているため，サービスに用いる IP の情報や，振り分け先のサーバの情報を，カーネル空間に伝える必要がある．IPVS では，Linux のカーネル空間とユーザ空間の通信を可能にする netlink を用いて情報を伝達をする．

IPVS の作者により，netlink を用いて情報の伝達をする部分を抽象化したライブラリである，libipvs が公開されている．ここでは，libipvs を用いて C 言語から IPVS を操作する例について述べる．

例として次のような状況を考える．これを実現する C 言語のコードを図 5 に示す．

- サービスは 10.0.0.1 という IP アドレスの TCP 80 番ポートを用いる．
- 振り分け先のサーバの 1 台目は 192.168.0.1 という IP アドレスの 80 番ポートを用いる．
- 振り分け先のサーバの 2 台目は 192.168.0.2 という IP アドレスの 80 番ポートを用いる．
- 振り分けのアルゴリズムは重み付きラウンドロビンを用いる．
  - 重みは振り分け先のサーバの 1 台目で 1 を用いる．
  - 重みは振り分け先のサーバの 2 台目で 3 を用いる．
- リクエストの振り分けは NAT (Network Address Translation) を用いる．

図 5 では，IPVS のサービスに用いる情報を，`ipvs_service_t` 構造体に保存している．

ipvs_service_t 構造体には，IP アドレス，アドレスファミリ，ポート，プロトコル，振り分けアルゴリズム名などを保存する．10 行目の parse_service 関数は，引数に情報を受け取り ipvs_service_t 構造体へ保存する．振り分け先サーバの情報は，ipvs_dest_t 構造体に保存する．ipvs_dest_t 構造体には，IP アドレス，アドレスファミリ，ポート，必要な場合は重みを保存する．23 行目の parse_dest 関数は，引数に情報を受け取り，ipvs_dest_t 構造体へ保存する．34 行目 からの main 関数では，ipvs_init 関数を呼び出し IPVS を使用する準備を行い，メモリの確保，構造体への書き込みを行った後，ipvs_add_service 関数を呼び出しサービスを IPVS へ登録をする．それぞれの振り分け先サーバに関しても同様に，ipvs_add_dest 関数を呼び出し振り分け先を IPVS へ登録をする．

```c
#include <ip_vs.h>
#include <libipvs.h>
#include <stdlib.h>
#include <string.h>

struct ipvs_service_t;
struct ipvs_dest_t;

// ipvs_service_t 構造体に，引数で与えられた情報を代入する関数
static void parse_service
(char *addr, int port, char *sched_name, ipvs_service_t *svc){
  struct in_addr inaddr;
  if (inet_aton(addr, &inaddr) != 0) {
    svc->addr.ip = inaddr.s_addr;
    svc->af = AF_INET;
    svc->port = htons(port);
    svc->protocol = IPPROTO_TCP;
    strcpy(svc->sched_name, sched_name);
  }
}

// ipvs_dest_t 構造体に，引数で与えられた情報を代入する関数
static void parse_dest
(char *addr, int port, int weight, ipvs_dest_t *dest){
  struct in_addr inaddr;
  if (inet_aton(addr, &inaddr) != 0) {
    dest->addr.ip = inaddr.s_addr;
    dest->af = AF_INET;
    dest->port = htons(port);
    dest->weight = weight;
  }
}

main()
{
  ipvs_service_t *svc;
  ipvs_dest_t *dest1, *dest2;
  // ipvs が存在するかどうか確認する
  if (ipvs_init())
    return 1;
  // ipvs_service_t, ipvs_dest_t 構造体のメモリ確保
  svc = (ipvs_service_t *)malloc(sizeof(ipvs_service_t));
  dest1 = (ipvs_dest_t *)malloc(sizeof(ipvs_dest_t));
  dest2 = (ipvs_dest_t *)malloc(sizeof(ipvs_dest_t));
  // 確保したメモリをクリアする
  memset(svc, 0, sizeof(ipvs_service_t));
  memset(dest1, 0, sizeof(ipvs_dest_t));
  memset(dest2, 0, sizeof(ipvs_dest_t));
  // 定義した関数を呼び出し，それぞれの構造体に情報を書き込む.
  parse_service("10.0.0.1", 80, "wrr", svc);
  parse_dest("192.168.0.1", 80, 1, dest1);
  parse_dest("192.168.0.2", 80, 3, dest2);
  // libipvs に定義されている関数を呼び出し，
  // 実際にサービス，振り分け先に追加する.
  ipvs_add_service(svc);
  ipvs_add_dest(svc, dest1);
  ipvs_add_dest(svc, dest2);
  return 0;
}
```

図 5　C 言語を用いた IPVS の操作例

3 既存の IPVS インタフェース

本章では，既存の IPVS インタフェースである Keepalived ならびに ipvsadm に関して説明し，それらの利用例と問題点について述べる．

3.1 keepalived

Keepalived は IPVS 単独では提供しないヘルスチェック機能や，仮想ルータ冗長プロトコル (Virtual Router Redundancy Protocol，以下 VRRP) を用いた，ロードバランサのフェイルオーバ機能を備えたソフトウェアである．Keepalived は設定に独自の DSL を用いる．

3.1.1 設定ファイル

2.2.3 節の状況を実現する Keepalived の設定ファイルは，図 6 のように書ける．

3.1.2 ヘルスチェック

Keepalived はヘルスチェック機能を提供する．Keepalived によるヘルスチェック機能を用いるための設定ファイルの一部を，図 7 に示す．

設定ファイルの `real_server` ブロック内でヘルスチェックの方式を指定する．使用できる方式の例を次に挙げる．

- `HTTP_GET`
  HTTP プロトコルの GET メソッドを用いて，ヘルスチェックを行う方式 (レイヤ 7 ヘルスチェック)．
- `TCP_CHECK`
  TCP ポートの応答があるかどうかでヘルスチェックを行う方式 (レイヤ 4 ヘルスチェック)．
- `MISC_CHECK`

```
1  virtual_server 10.0.0.1 80 {     ! 仮想 IP 10.0.0.1 ポート80番の定義
2    lb_algo wrr                    ! アルゴリズムは重み付きラウンドロビン
3    lb_kind NAT                    ! パケットの転送方法は NAT
4    protocol TCP                   ! プロトコルは TCP
5
6    real_server 192.168.0.1 80 {   ! 実サーバ 192.168.0.1 ポート80番の定義
7      weight 1                     ! 重み 1
8    }
9    real_server 192.168.0.2 80 {   ! 実サーバ 192.168.0.2 ポート80番の定義
10     weight 3                     ! 重み 3
11   }
12 }
```

図 6　Keepalived を用いた IPVS の操作例

```
 1 real_server 192.168.0.1 80 {
 2   HTTP_GET {                  ! ヘルスチェックの方法を指定
 3     url {                     !
 4       path /                  ! HTTP メソッドで GET するパスを指定
 5       status_code 200         ! OK とするステータスコードを指定
 6     }
 7     connect_timeout 10        ! チェックのタイムアウト秒数
 8     nb_get_retry 3            ! チェックに失敗した時のリトライ回数
 9     delay_before_retry 5      ! リトライ時に待機する秒数
10   }
11 }
```

図 7　Keepalived を用いたヘルスチェックの例

指定したパスのスクリプトを実行し, その終了コードによってヘルスチェックを行う方式.

3.1.3　ロードバランサの冗長化

Keepalived は，VRRP を用いてロードバランサ自体の冗長化を行う機能も備えている．ネットワーク内に存在する VRRP に対応している他のロードバランサと一定の時間間隔で通信を行い，互いに障害が発生していないか確認をする．ロードバランサ間には優先度が設定されており，通常，最も高い優先度のロードバランサが仮想 IP アドレスを持つ．自分より優先度が高いロードバランサから VRRP 応答が無くなった場合，障害が発生したと判断し，2 番目に優先度の高いロードバランサが自動で仮想 IP アドレスを持つようになる．これにより，ロードバランサの障害が発生した際にも，僅かなダウンタイムで自動復旧することが可能である．

3.1.4　問題点

Keepalived では, 現在の設定値を取得し，制御構文を用いて条件分岐を行うことや DSL に新たな文法を追加するなどの拡張を行うことが困難である．

また，設定ファイルの構文チェックが無く，誤った文法で設定を書いてしまった場合，意図しない設定で動作を開始し，サービスに深刻な影響を与える危険性がある．

例えば，図 8 のように，8 行目の閉じ中括弧が欠落した場合を考える．Keepalived は 6 行目から 11 行目を 1 つのブロックとして扱う．9 行目は意味のない構文として無視され，10 行目は 7 行目の設定を上書きする．つまり，図 8 は，仮想 IP 10.0.0.1 の 80 番ポートに対し，192.168.0.1 の 80 番ポートを重み 3 の実サーバとして登録し，192.168.0.2 の 80 番ポートは登録されない．

ヘルスチェックにおける，MISC_CHECK は外部スクリプトの実行を行う．MISC_CHECK に正しく終了しないスクリプトを指定した場合は，ロードバランサ自体のシステム障害を引き起こす可能性がある．また，あくまで外部スクリプトの実行であるため，現在の振り分け状況や，接続数などのロードバランサの状態に応じた振り分けを外部スクリプトを用いて行うのは困難である．

```
 1 virtual_server 10.0.0.1 80 {
 2   lb_algo wrr
 3   lb_kind NAT
 4   protocol TCP
 5
 6   real_server 192.168.0.1 80 {  ! 6行目から11行目が，
 7     weight 1                    ! 1つのブロックとして扱われる
 8                                 !
 9   real_server 192.168.0.2 80 {  ! 9行目は意味のない構文として無視される
10     weight 3                    ! 10行目は7行目の設定を上書きする
11   }                             !
12 }
```

図 8　Keepalived の設定ミスの例

```
1 #!/bin/bash
2 # アルゴリズムは重み付きラウンドロビン
3 # サービス 10.0.0.1 ポート 80番のサービスを追加
4 ipvsadm -A -t 10.0.0.1:80 -s wrr
5 # 10.0.0.1:80 のサービスに対し，192.168.0.1:80 を NAT かつ 重み 1 で追加
6 ipvsadm -a -t 10.0.0.1:80 -r 192.168.0.1:80 -m -w 1
7 # 10.0.0.1:80 のサービスに対し，192.168.0.2:80 を NAT かつ 重み 1 で追加
8 ipvsadm -a -t 10.0.0.1:80 -r 192.168.0.2:80 -m -w 1
```

図 9　ipvsadm を用いた IPVS の操作例

3.2　ipvsadm

ipvsadm は，IPVS を Linux 上で操作するためのインタフェースコマンドである．

3.2.1　利用例

2.2.3 節の状況を実現するコードは，図 9 のように書ける．

ipvsadm はコマンドライン上から利用することが可能なため，シェルスクリプトへの組み込みや，コマンド実行が可能なスクリプト言語と併用することが出来る．

3.2.2　問題点

その反面，ipvsadm を用いてヘルスチェック機構や動的な振り分け先の追加などを行う場合，現在の振り分けに関する関する情報を取得することが困難である．

例えば，ipvsadm を用いて，現在のサービスと振り分け状況を取得するコマンドを実行した場合，図 10 のように印字される．

192.168.0.1 の 80 番ポートの重みがいくつであるかを知るには，Linux 上の grep コマンドを用いれば図 11 のように書くことができる．DEST1 や DEST1_WEIGHT といった変数はコマンド実行時の値がそのまま入る．そのため，重みを変更した場合は変数の値と実際の重みに不整合が生じる．

```
[root@localhost ~]# ipvsadm -L
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  10.0.0.1:http wrr
  -> 192.168.0.1:http             Masq    1      0          0
  -> 192.168.0.2:http             Masq    1      0          0
```

図 10　ipvsadm コマンドを用いた振り分け状況の表示

```
1 #!/bin/bash
2 # ipvsadm -L コマンドの結果から 192.168.0.1 に関する行を検索し
3 # DEST1 という変数に配列として格納
4 DEST1=(`ipvsadm -Ln | grep 192.168.0.1`)
5 # 配列の要素はスペース区切りで入るため，4 つ目の要素が重みに該当する
6 DEST1_WEIGHT=${DEST1[3]}
```

図 11　ipvsadm を用いた重みの取得

```
[root@localhost ~]# ipvsadm -Ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  10.0.0.1:80 wrr
  -> 192.168.0.1:80               Masq    1      0          0
  -> 192.168.0.2:80               Masq    1      0          0
TCP  10.0.0.2:80 wrr
  -> 192.168.0.1:80               Masq    2      0          0
  -> 192.168.0.3:80               Masq    1      0          0
UDP  10.0.0.1:53 wrr
  -> 192.168.0.1:53               Masq    2      0          0
  -> 192.168.0.2:53               Masq    3      0          0
  -> 192.168.0.3:53               Masq    4      0          0
```

図 12　設定情報の取得が困難な出力例

さらに，振り分け先がいくつあるか分からない場合や，複数のサービスに同じ振り分け先が登録されていた場合，どのサービスで使われている振り分け先の重みなのか分からないなどの問題が発生するため，実用的ではない．重みなどの設定情報の取得が困難であるような ipvsadm コマンドの出力例を図 12 に示す．

4 設計

本章では，本システムの設計について述べる．実装対象の言語ならびに実装方針，実装後の利用例に関して説明する．

4.1 mruby

本システムでは，一般に用いられている Ruby の 組み込み向け軽量版である mruby [3] から IPVS を操作できる機構を実装する．mruby は，基本的な言語機能や文法は CRuby の JIS 規格を強く意識している．そのため，CRuby とほぼ同等の記述性や生産性を持っている．また，組み込み向けであるため，次のような利点がある．

- C 言語で作られたアプリケーションとの親和性が高いこと．
- 言語自体のサイズが小さいこと．
- メモリ使用量が大きくならない配慮がなされていること．
- リアルタイム処理に配慮がなされていること．

そのため，mruby は本システムのようにサーバ上で長時間動作し，ソフトリアルタイム性が求められるシステムに適した言語である．

mruby には拡張ライブラリを言語内に組み込む mrbgem という機構が存在する．mrbgem は，C 言語で書かれた拡張ライブラリを mruby のビルド時に組み込むことを可能にするものであり，mruby から 既存の C 言語アプリケーションを操作する際によく用いられる手法である．本システムでは，IPVS に関する構造体や関数呼び出しといった部分を C 言語で記述し，それを mruby 上から呼び出せるような mrbgem を作成する．

4.2 クラス構造

名前空間として IPVS という名前のモジュールを定義する．その後，libipvs の構造体と mruby のクラスを対応付け，libipvs に定義されている関数を，mruby 上の適切なクラスのメソッドとして定義する．

libipvs で主に用いられている構造体は，次の 4 つである．

- `ipvs_service_t`
  サービスに関する情報を保存する構造体
- `ipvs_dest_t`
  振り分け先に関する情報を保存する構造体
- `ipvs_timeout_t`
  接続のタイムアウト値に関する情報を保存する構造体
- `ipvs_daemon_t`
  syncmaster に関する情報を保存する構造体

```
1  # Ruby では インスタンス を new したときに,
2  # initialize という名前のインスタンスメソッドが実行される.
3  s = IPVS::Service.new({
4    'addr' => '10.0.0.1',
5    'port' => 80,
6    'sched_name' => 'wrr'
7  }).add_service
8
9  # 振り分け先の定義
10 d1 = IPVS::Dest.new({
11   'addr' => '192.168.0.1',
12   'port' => 80,
13   'weight' => 1
14 })
15
16 d2 = IPVS::Dest.new({
17   'addr' => '192.168.0.2',
18   'port' => 80,
19   'weight' => 3
20 })
21
22 # 振り分け先をサービスに追加
23 s.add_dest(d1)
24 s.add_dest(d2)
25
26 s.addr #=> 10.0.0.1
27 s.port #=> 80
28 d1.weight #=> 1
29 d1.weight = 3 #=> nil
30 # d1.weight=(3) と書いてもよい
31 d1.weight #=> 3
```

図 13　本システムの利用例

本システムでは，以下の 2 つのクラスを定義する．

- Service クラス
  ipvs_service_t 構造体と対応している．
- Dest クラス
  ipvs_dest_t 構造体に対応している．

残りの ipvs_timeout_t 構造体ならびに，ipvs_daemon_t 構造体は，IPVS 全体に関係するため，IPVS モジュールのメソッドで操作を可能にする．

4.3　本システムの利用例

本システムの利用例を図 13 に示す．本システムでは，新しくサービスを定義したい場合には，まず IPVS モジュール下の Service クラスのインスタンスを生成する．mruby では CRuby と同様，インスタンスを生成のために new メソッドを実行する．その際，initialize という名前のインスタンスメソッドが実行される．Service クラスの initialize メソッドは，Hash ク

ラスのオブジェクトを 1 つ引数に取る (3 行目から 7 行目) . 渡されるオブジェクトは，addr，port，sched_name などの，サービスの定義に必要なキーと値のペアを持っている必要がある．ここで指定可能なキーは，次の通りである．

- addr
  IP アドレスを指定する．IP アドレスのあとにコロンをつけてポート番号を指定することも可能である．
- port
  ポート番号を指定する．デフォルト値は 0．addr と port で同時に値を指定した場合，port で指定したものが優先される．
- protocol
  プロトコルを指定する．デフォルト値は TCP.
- sched_name
  振り分けアルゴリズムを指定する．デフォルト値は 重み付き最小接続数.
- timeout
  セッションを維持する秒数を指定する．デフォルト値は 0.

このうち addr は必須であるが，他のものは省略可能であり，省略した場合にはデフォルト値を使用する．Service クラスのインスタンスを生成しただけでは，実際に IPVS へ追加は行われない．add_service メソッドを呼び出した際に，IPVS へサービスが追加される (5 行目) . IPVS からサービスを削除したい場合は，del_service メソッドを呼び出す.

また，実サーバを定義する場合は，IPVS モジュールに定義されている Dest クラスのインスタンスを生成する．Dest クラスの initialize メソッドも，Service クラスと同様の引数の受け取り方をする．ここで指定可能なキーは，次の通りである．

- addr
  IP アドレスを指定する．サービスの時と同様，IP アドレスのあとにコロンをつけてポート番号を指定することも可能である．
- port
  ポート番号を指定する．デフォルト値は 0．addr と port で同時に値を指定した場合，port で指定したものが優先される．
- weight
  振り分け先の重みを指定する．デフォルト値は 1.

Dest クラスもインスタンスを生成しただけでは IPVS に実サーバとしては追加されない．Service クラスのインスタンスメソッド add_dest の引数に，Dest クラスのインスタンスを渡すことで，レシーバのサービスの振り分け先に実際に追加される (23，24 行目) .

振り分けから削除する際には，Service クラスのインスタンスメソッドである del_dest を，追加したサービスのインスタンスから呼び出し，削除する Dest クラスのインスタンスを引数に渡す.

`Service` クラスならびに `Dest` クラスのインスタンスは共に，それぞれの設定値と同名のメソッドが定義してあり，

そのメソッドを呼び出すことにより，設定値を確認することが可能である (26 行目から 28 行目) ．また，設定値にイコールをつけたメソッドに引数を渡すことにより，設定値を変更することが可能である (29 行目) ．サービスや実サーバが IPVS に登録済みだった場合，値を変更すると即座に IPVS に反映される．

5 実装

本章では，本システムの実装について述べる．主に mruby でのクラス実装や，メソッド実装に関して，実際のコードを例に取り詳細を述べる．

5.1 mrbgem の定義

mrbgem では，作成するライブラリの定義を行う必要がある．mrbgem のライブラリの定義は，作成するライブラリのルートディレクトリの `mrbgem.rake` という名称のファイルによって行う．ここでは，ライブラリの名前を mruby-ipvs として定義を行うため，`mrbgem.rake` ファイルを図 14 のように記述した．

1 行目で ライブラリの名前が mruby-ipvs であることを定義している．2，3 行目では ライブラリのライセンス，作成者の名前を定義している．4 行目から 9 行目では，依存関係にあるライブラリや，インクルードパスなどの，ライブラリをコンパイルするために必要な情報を定義している．本システムでは，libipvs を利用し，libipvs が netlink に依存しているため，5 行目で依存関係にあるライブラリとして netlink を指定し，libipvs を利用するために 4 行目で libipvs のヘッダファイルをインクルードパスに指定し，7 行目から 9 行目で libipvs のオブジェクトファイルを作成し，利用できるようにしている．

5.2 クラスとメソッドの定義

mrbgem では，作成するライブラリ名によって決定される初期化関数が最初に実行される．ライブラリの名前を mruby-ipvs として定義したため，初期化関数の名前は `mrb_mruby_ipvs_gem_init` となる．そのため，`mrb_mruby_ipvs_gem_init` 関数を定義し，その中でクラスとメソッドの定義を行う．`mrb_mruby_ipvs_gem_init` 関数の一部を図 15 に記す．

mrbgem の 初期化関数では，mruby の状態を保持した `mrb_state` 型の変数が引数として渡される．2 行目から 4 行目では，モジュールならびにクラスとして用いる構造体を定義している．7 行目で，`mrb_define_module` 関数により，`IPVS` モジュール を定義している．

```
 1 MRuby::Gem::Specification.new('mruby-ipvs') do |spec|
 2   spec.license = 'MIT'
 3   spec.author  = 'YOSHIKAWA Ryota'
 4   spec.cc.include_paths << "#{spec.dir}/libipvs-2.6"
 5   spec.linker.libraries << ['nl']
 6   spec.objs << (Dir.glob("#{spec.dir}/src/*.c") +
 7                 Dir.glob("#{spec.dir}/libipvs-2.6/*.c")).map { |f|
 8     f.relative_path_from(spec.dir).pathmap("#{build_dir}/%X.o")
 9   }
10 end
```

図 14　mrbgem.rake

```
 1 void mrb_mruby_ipvs_gem_init(mrb_state *mrb) {
 2   struct RClass* _module_ipvs;
 3   struct RClass* _class_ipvs_service;
 4   struct RClass* _class_ipvs_dest;
 5
 6   // IPVS モジュールの定義
 7   _module_ipvs = mrb_define_module(mrb, "IPVS");
 8
 9   // IPVS::Service
10   // IPVS モジュール配下に Service クラスを定義する
11   _class_ipvs_service = mrb_define_class_under(mrb, _module_ipvs,
12       "Service", mrb->object_class);
13   // Service クラスの initializer を定義する
14   mrb_define_method(mrb, _class_ipvs_service,
15       "initialize", mrb_ipvs_service_init, ARGS_REQ(1));
16   mrb_define_method(mrb, _class_ipvs_service,
17       "initialize_copy", mrb_ipvs_service_init_copy,
18       ARGS_REQ(1));
19   // Service クラスにインスタンスメソッドを定義する
20   mrb_define_method(mrb, _class_ipvs_service,
21       "add_service", mrb_ipvs_service_add, ARGS_NONE());
22   mrb_define_method(mrb, _class_ipvs_service,
23       "del_service", mrb_ipvs_service_del, ARGS_NONE());
24   .
25   .
26   .
27   // IPVS::Dest
28   // IPVS モジュール配下に Dest クラスを定義する
29   _class_ipvs_dest = mrb_define_class_under(mrb, _module_ipvs,
30       "Dest", mrb->object_class);
31   // Dest クラスの initializer を定義する
32   mrb_define_method(mrb, _class_ipvs_dest,
33       "initialize", mrb_ipvs_dest_init, ARGS_REQ(1));
34   mrb_define_method(mrb, _class_ipvs_service,
35       "initialize_copy", mrb_ipvs_service_init_copy,
36       ARGS_REQ(1));
37   // Dest クラスにインスタンスメソッドを定義する
38   mrb_define_method(mrb, _class_ipvs_dest,
39       "addr", mrb_ipvs_service_get_addr, ARGS_NONE());
40   mrb_define_method(mrb, _class_ipvs_dest,
41       "port", mrb_ipvs_service_get_port, ARGS_NONE());
42   .
43   .
44   .
45 }
```

図 15　mrb_mruby_ipvs_gem_init 関数の一部

mrb_define_module 関数は，第 1 引数に mruby の状態変数，第 2 引数にモジュール名を渡すことによって，モジュールを定義することが出来る．

11 行目で，mrb_define_class_under 関数により，IPVS モジュールの下位に，Service クラスを定義している．mrb_define_class_under 関数は，第 2 引数にクラスまたはモジュールを渡すことにより，その下位に，第 3 引数で渡した名称のクラスが定義される．第 4 引数では，定義するクラスの型を指定している．

14 行目から 23 行目では，Service クラスに対してインスタンスメソッドを定義している．インスタンスメソッドの定義は，mrb_define_method で行う．第 2 引数にそのメソッドを定義するクラスを，第 3 引数にメソッド名を指定する．第 4 引数は，メソッドが呼び出された際に実行される C 言語の関数を指定する．第 5 引数は，メソッドの引数に関する情報を指定し，必須である場合は ARGS_REQ 句に必須である引数の個数を与える．省略可能な引数を指定したい場合は ARGS_OPT 句の引数に，いくつまで受け入れるかを記述する．引数を受け取らない場合は，ARGS_NONE() を記述する．また，mruby では，initialize という名前のインスタンスメソッドを定義した場合，インスタンスを生成した際に実行されるようになる．

5.3 構造体とクラスの対応付け

mruby では，C 言語の構造体をラップし，mruby のオブジェクトとして扱うことが可能である．構造体のラップは，クラスの初期化関数で行われる．例として，Service クラスの初期化関数の一部を図 16 に示す．

図 16 では，1 行目から 3 行目で，Service クラスのデータタイプを設定している．mrb_data_type 構造体は，1 つ目のメンバに，クラスとしてラップする対象の構造体名を指定し，2 つ目のメンバに，生成されたオブジェクトがごみとなった時に呼び出される関数を指定する．ここでは，mruby のメモリ管理 API である mrb_free を指定している．

8 行目から 12 行目では，mruby から渡される引数を受け取るための変数を宣言している．

19 行目では，メソッド引数を受け取る mrb_get_args 関数を呼び出している．mrb_get_args 関数は，第 2 引数で受け取れるオブジェクトのクラスを指定でき，第 3 引数で指定したオブジェクトに代入する．なお，"H" は Hash クラスを示している．

24 行目では，mrb_hash_opt 関数を用いて，19 行目で代入した arg_opt から，key が addr であるものを取り出し，addr 変数に代入している．mrb_hash_opt 関数の第 2 引数には対象の Hash を指定する．第 3 引数では，mruby の String クラスのオブジェクトでキーを指定する必要がある．そのため，mrb_str_new_cstr 関数という String クラスのオブジェクトを生成する関数を呼び出し，渡している．

38 行目では，生成したオブジェクトのデータタイプが何であるか指定している．ここでは，1 行目から 3 行目で設定した構造体を指定している．

42 行目では，生成されたオブジェクトのポインタを指定しており，初期化関数で生成した ipvs_service_t 構造体のポインタを指定している．これを指定しておくことによって，インスタンスメソッドから構造体のポインタを参照することが可能になる．

5.4 メソッドの定義

mruby におけるメソッドの定義について述べる．例として，Service クラスのインスタンスメソッドの定義のいくつかを図 17 に示す．

mrb_ipvs_service_get_port 関数は，Service クラスのインスタンスメソッドである port メソッドと対応している．前節で述べた通り，インスタンスの initialize メソッドを呼び出

```
 1 const static struct mrb_data_type mrb_ipvs_service_type = {
 2   "Service", mrb_free
 3 };
 4 static mrb_value
 5 mrb_ipvs_service_init(mrb_state *mrb, mrb_value self){
 6   int parse;
 7   // mruby で用いるオブジェクトの変数宣言
 8   mrb_value arg_opt = mrb_nil_value(),
 9             addr = mrb_nil_value(),
10             ...
11   // mruby で用いる整数型の変数宣言
12   mrb_int port, timeout, netmask;
13   struct ipvs_service_t *svc;
14
15   svc = (struct ipvs_service_t*)mrb_malloc(mrb, sizeof(*svc));
16   memset(svc, 0, sizeof(struct ipvs_service_t));
17
18   // メソッド引数の Hash を受け取る
19   mrb_get_args(mrb, "H", &arg_opt);
20   if (mrb_nil_p(arg_opt))
21     mrb_raise(mrb, E_ARGUMENT_ERROR, "invalid argument");
22
23   // ハッシュから key が "addr" の value を取り出し，addr 変数に保存する
24   addr = mrb_hash_get(mrb, arg_opt, mrb_str_new_cstr(mrb, "addr"));
25   if (mrb_nil_p(addr))
26     mrb_raise(mrb, E_ARGUMENT_ERROR, "invalid argument");
27
28   ...
29
30   // メソッドの引数から受け取った addr のポインタを取り出し，
31   // ipvs_service_t 構造体へ書き込むための関数へ渡す
32   parse = parse_service((char *) RSTRING_PTR(addr), &svc);
33
34   ...
35
36   // オブジェクトのデータタイプを，
37   // mrb_ipvs_service_type に設定する
38   DATA_TYPE(self) = &mrb_ipvs_service_type;
39
40   // 生成されたオブジェクトのポインタが，
41   // svc を参照するように設定する
42   DATA_PTR(self) = svc;
43
44   // 生成したオブジェクトを返却する
45   return self;
46 }
```

図 16　mrb_ipvs_service_init 関数の一部

した際に実行される関数により，インスタンスと対応している ipvs_service_t 構造体のポインタが参照できるようになっているので，DATA_PTR マクロを用いて，構造体のポインタを獲得している (5 行目)．獲得した構造体から，保存されているポート番号を取り出し，ネットワークバイトオーダからホストバイトオーダに変換したものを，mruby の Fixnum オブジェクトのインスタンスとして返却している．

```
 1 static mrb_value
 2 mrb_ipvs_service_get_port(mrb_state *mrb, mrb_value self){
 3   struct ipvs_service_t *svc;
 4   // Service クラスのインスタンスのに設定されているポインタを代入
 5   svc = DATA_PTR(self);
 6   // ポート番号を mruby の整数型に変換して返却
 7   return mrb_fixnum_value(ntohs(svc->port));
 8 }
 9
10 static mrb_value
11 mrb_ipvs_service_add(mrb_state *mrb, mrb_value self){
12   errno = 0;
13   // Service クラスのインスタンスに設定されているポインタを,
14   // IPVS のサービスを追加するための関数に渡す
15   ipvs_add_service(DATA_PTR(self));
16   if (errno)
17     mrb_raise(mrb, E_RUNTIME_ERROR, ipvs_strerror(errno));
18   return mrb_nil_value();
19 }
20
21 // Service クラスのインスタンスメソッド
22 // サービスに 振り分け先を追加する add_dest メソッドに対応している
23 // add_dest() メソッドは引数に Dest クラスのインスタンスを要求する
24 static mrb_value
25 mrb_ipvs_service_add_dest(mrb_state *mrb, mrb_value self){
26   mrb_value arg;
27   errno = 0;
28
29   mrb_get_args(mrb, "o", &arg);
30   // 引数のオブジェクトのデータタイプが,
31   // Dest クラスで設定されたものかどうかを判定
32   if(!(DATA_TYPE(arg) == &mrb_ipvs_dest_type))
33     mrb_raise(mrb, E_ARGUMENT_ERROR, "invalid argument");
34   // IPVS の振り分け先追加用関数に,
35   // 第1引数として Service クラスのインスタンスで設定されたポインタ,
36   // 第2引数として Destクラスのインスタンスで設定されたポインタを渡す
37   ipvs_add_dest(DATA_PTR(self), DATA_PTR(arg));
38   if (errno)
39     mrb_raise(mrb, E_RUNTIME_ERROR, ipvs_strerror(errno));
40   return mrb_nil_value();
41 }
```

図 17　mruby のメソッド定義の一部

mrb_ipvs_service_add 関数は, Service クラスのインスタンスメソッドである add_service メソッドと対応している．こちらも DATA_PTR を用いてインスタンスと対応している構造体のポインタを取り出し，libipvs にて定義されている，ipvs_add_service 関数に渡している．何らかのエラーが発生した場合には，errno に値が保存される．保存された errno を，libipvs で定義されている ipvs_strerror 関数に渡すことで，char * 型でエラー内容を受け取る事ができる．

例えば, 既に追加されているものと同様のサービスを追加しようとした場合, ”Service already exists” というメッセージを出力し，ランタイムエラーを発生させる (16，17 行目) .

mrb_ipvs_service_add_dest 関数は，Service クラスのインスタンスメソッドである

add_dest メソッドと対応している．add_dest メソッドは，引数として Dest クラスのインスタンスを受け取り，受け取ったインスタンスを実サーバとして登録するものである．Dest クラスに関しても Service と同様，initialize メソッドと対応している関数内で，データタイプを指定し，構造体のポインタを参照できるようにしている．そのため，受け取った引数が Dest クラスのインスタンスであるかの判定は，34 行目のように書くことが可能である．その後 Service クラスのインスタンス自体を表す self と，引数である arg からそれぞれ構造体のポインタを取り出し，ipvs_add_dest 関数に渡すことによって，実サーバへの追加を行っている．

6 評価

本章では，本システムの評価に関して述べる．評価は，記述性の評価と，ベンチマークによる評価に分かれる．記述性の評価に関しては，本システムを用いて記述した簡単なプログラム関して考察する．

6.1 記述性の評価

本システムを使って，外部の mrbgem を使用しながら IPVS の操作を行う例と，Keepalived の DSL を模倣したものを作成した例を示し，そのそれぞれにおいて，既存のインタフェースと比較し，優位点を述べる．

6.1.1 mrbgem を利用した IPVS の操作

サードパーティの mrbgem である mruby-curl [4] ，mruby-process [5] ならびに mruby-sleep [6] を用いて Web サーバのヘルスチェックを行いながら，振り分けを制御する例を図 18 に示す．

3 行目から 8 行目に関しては，サービスならびに実サーバのインスタンスを生成している．8 行目の `sorry` に関しては，全ての Web サーバに障害が発生した際に，メンテナンスページを表示するためのサーバとして定義しており，すぐには `add_dest` しない．

13 行目からがヘルスチェックを行っている部分であり，29 行目の mruby-sleep を用いた `sleep 5` があるため，最後のヘルスチェックが終了してから 5 秒後に次のヘルスチェックが実行される．

14 行目から 19 行目に関しては，1 台目の Web サーバである `web1` のヘルスチェックを行っている．14 行目で，mruby-curl を用いて，HTTP の GET メソッドを使って，レイヤ 7 のヘルスチェックを行っている．レスポンスボディが `nil` であった場合，つまり正常なレスポンスを返していなかった場合，16 行目に進む．16 行目では，`web1` サーバのインアクティブな接続の数とアクティブな接続の数を確認し，インアクティブな接続がアクティブなコネクションを大きく上回っていた場合，mruby-xquote を用いて，`web1` サーバに ssh ログインを行い，Web サーバである httpd の再起動を行っている．22 行目では，正しくレスポンスボディが返され，かつ現在の振り分け先に `web1` がなかった場合，再度 `web1` を振り分け先に追加する処理を行っている．25 行目から 30 行目は `web2` に関して同様の処理を行っている．33 行目では，振り分け先の全てのサーバで障害が発生し，振り分け先が無くなった場合に，8 行目で定義した `sorry` 変数を振り分け先に追加し，メンテナンスページの表示などを行うようにしている．

この例における，Keepalived や ipvsadm を用いたスクリプティングとの差分として，振り分け先へのコネクションやアドレス，重みなどを容易に取得でき，それを元に振り分けを変更できる点が挙げられる．また，mruby 特有の `include?` メソッドなどが使えることで，より簡潔に記述出来る点も挙げられる．また，Keepalived の `MISC_CHECK` と違い，確実に終了するかなどに気を使わなくて良いため，16 行目のように Web サーバの再起動をスクリプト内で行える点も

```
# サービスを定義
# port は IP アドレスに : をつけて繋げることでも指定可能
sercive = IPVS::Service.new({
  'addr' => '10.0.0.1:80', 'sched_name' => 'wrr'
}).add_service
web1 = IPVS::Dest.new({'addr' => '192.168.0.1:80'})
web2 = IPVS::Dest.new({'addr' => '192.168.0.2:80'})
sorry = IPVS::Dest.new({'addr' => '192.168.0.3:80'})

service.add_dest(web1)
service.add_dest(web2)

loop do
  if Curl::get(web1.addr).body == nil
    # web1 の接続数の確認
    if web1.inactconns > 500 && web1.activeconns < 50
      # web1 に ssh ログインして httpd プロセスを再起動
      `ssh #{web1.addr} -t sudo /etc/init.d/httpd restart`
    end
  else
    # ヘルスチェックに成功し，振り分け先に web1 が無ければ追加
    service.add_dest(web1) unless sercive.dests.include? web1
  end
  # web2 のヘルスチェック
  if Curl::get(web2.addr).body == nil
    service.del_dest(web2)
  else
    # ヘルスチェックに成功し,振り分け先にweb2が無ければ追加
    service.add_dest(web2) unless sercive.dests.include? web2
  end
  # 振り分け先が無くなった場合，sorry サーバを追加
  if service.dests == nil
    service.add_dest(sorry)
  end
  sleep 5
end
```

図 18　ヘルスチェックと振り分け制御を行う例

挙げられる．

ただし，この例のようなプログラムを実際に運用した場合，振り分け台数の増加や，サービスの増加に伴い，ヘルスチェックに掛かる時間が大幅に伸びてしまい，ダウンタイムが非常に長くなる危険性が考えられる．解決策として，mruby でスレッドを扱えるようにした mruby-thread を用いるなどして，ヘルスチェックを並列化するなどの処理を記述することが考えられる．このように，適切なライブラリを言語に組み込むことにより，問題を解決することができるため，既存のインタフェースに比べて優位である．

6.1.2　Keepalived の DSL

mruby の記述性を生かし，Keepalived の DSL の構文と似た形で，IPVS を操作できる例を示す．

まず，Keepalived の DSL と似た構文を実装するためのプログラムを，図 19 に示す．また，

```
 1 module Keepalived
 2   # モジュール内で使う変数の定義
 3   @lb_algo = "wrr", @lb_kind = "NAT", @protocol = "TCP"
 4   @weight = 1, @real_servers = [], @virtual_server = nil
 5
 6   # 変数の設定用メソッド
 7   def self.lb_algo(algo)
 8     @lb_algo = algo
 9   end
10
11   def self.lb_kind(kind)
12     @lb_kind = kind
13   end
14
15   def self.protocol(proto)
16     @protocol = proto
17   end
18
19   def self.weight(weight)
20     @weight = weight
21   end
22
23   def self.virtual_server(addr, &block)
24     block.call # 第 2 引数として受け取ったブロックの実行
25     # 引数として受け取った addr の サービスを定義
26     @virtual_server = IPVS::Service.new({
27        "addr" => addr, "sched_name" => @lb_algo, "proto" => @protocol,
28     })
29
30     @virtual_server.add_service
31     # 振り分け先をサービスに追加
32     @real_servers.each do |real_server|
33       @virtual_server.add_dest(real_server)
34     end
35   end
36
37   def self.real_server(addr, &block)
38     block.call # 第 2 引数として受け取ったブロックの実行
39     # real_servers という名称の配列に，振り分け先の定義を代入
40     @real_servers << IPVS::Dest.new({
41       "addr" => addr, "weight" => @weight, "fwd" => @lb_kind
42     })
43   end
44 end
```

図 19　Keepalived の DSL の実現する Keepalived モジュール

図 19 のプログラムを用いて，Keepalived と似た構文により設定を行ったものを図 21 に示す．

図 19 では，Keepalived モジュールの定義を行っている Keepalived モジュールでは，サービスの定義や振り分け先の定義に用いる変数を保持できるようにしている (3 行目から 21 行目)．virtual_server メソッドならびに real_server メソッドでは，まず，受け取ったブロックを実行し，その後に Service もしくは Dest クラスのインスタンスを生成している (23 行目から 43 行目)．mruby の Kernel モジュールへの変更を行っている．図 20 では，Kernel モジュールへの追加として，virtual_server ， real_server，各種設定値を代入するメソッド

```
# Kernel モジュールに定義されたメソッドは，
# 全てのクラスから参照することが可能．
module Kernel
  # virtual_server というメソッドを定義
  # IP アドレスとブロックを受け取り
  # Keepalived モジュールの virtual_server メソッドへ受け渡す
  def virtual_server(addr, &block)
    ::Keepalived.virtual_server addr, &block
  end
  # 変数の定義用メソッドを呼び出す
  def lb_algo(algo)
    ::Keepalived.lb_algo algo
  end
  def lb_kind(kind)
    ::Keepalived.lb_kind kind
  end
  def protocol(proto)
    ::Keepalived.protocol proto
  end
  # real_server というメソッドを定義
  # IP アドレスとブロックを受け取り
  # Keepalived モジュールの real_server メソッドへ受け渡す
  def real_server(addr, &block)
    ::Keepalived.real_server addr, &block
  end
  # 変数の定義用メソッドを呼び出す
  def weight(weight)
    ::Keepalived.weight weight
  end
end
```

図 20　Keepalived の DSL を実現する Kernel モジュール

を定義している．virtual_server，real_server メソッドは 第 3 引数にブロックを受け取り，Keepalived モジュールの同名メソッドに受け渡す．各種設定値を代入するメソッドは，受け取った引数をそのまま Keepalived モジュールの同名メソッドに受け渡す．

実際の挙動に関しては，利用例である図 21 と共に解説する．図 21 では，図 20 において定義した，Kernel モジュールのメソッドを主に用いる．

1 行目は，Kernel::virtual_server メソッドの呼び出しであり，第 1 引数に addr，第 2 引数にブロックを渡している．

Kernel::virtual_server メソッドは，Keepalived::virtual_server メソッドを同じ引数で呼び出す．

呼び出された Keepalived::virtual_server メソッドは，まず 第 2 引数のブロックを実行する．図 21 の例では，Kernel::lb_algo，Kernel::lb_kind，Kernel::protocol メソッドが呼ばれ，これらは Keepalived モジュールの同名メソッドへ引き渡され，変数へセットされる．また，Kernel::real_server メソッドも呼ばれるが，これも Kernel::virtual_server とほぼ同様の振る舞いをする．

このように，mruby の記述性の高さを生かし，mruby-ipvs を用いた，独自の DSL のようなものを簡単に定義することが可能である．また，元は mruby スクリプトであるため，独自 DSL で

```
# 仮想 IP 10.0.0.1 80 番ポートのサービスを定義
virtual_server "10.0.0.1:80" do
  lb_algo "wrr"  # 振り分けアルゴリズムは 重み付きラウンドロビン
  lb_kind "NAT"  # パケット転送方式は NAT
  protocol "TCP" # プロトコルは TCP

  real_server "192.168.0.1:80" do # 192.168.0.1 80番ポートの
    weight 1                      # 振り分け先を 重み 1 で定義
  end

  real_server "192.168.0.2:80" do # 192.168.0.2 80番ポートの
    weight 2                      # 振り分け先を 重み 2 で定義
  end

end
```

図 21　Keepalived の DSL に似た構文の設定

ありながら制御構文を利用でき，かつ構文チェック機構も備えることができる点が，Keepalived より優っている．

6.2　ベンチマークによる評価

本研究では，既存のインタフェースより速度を向上させることを特に目的としていないが，本システムによる IPVS の操作速度の既存のシステムと同程度であることを本節で示す．

次のような環境を用いてベンチマークを実行した．

- OS
  CentOS release 6.4 (Final)
- CPU
  Intel(R) Core(TM) i5-4258U CPU @ 2.40 GHz 1 コア
- メモリ
  512 MiB

ここでは，ipvsadm を用いた操作と本システムを用いた操作について比較する．

IPVS に対するサービスと，実サーバの追加ならびに削除を 1,000 回連続して行った．サービスと実サーバの概要は次の通りである．

- サービス
  仮想 IP: 10.0.0.1
  プロトコル: TCP
  ポート番号: 80
  振り分けアルゴリズム: 重み付きラウンドロビン
- 実サーバ
  IP アドレス: 192.168.0.1

```
#!/bin/bash
# 仮想 IP アドレス 10.0.0.1 の 80 番ポート,
# 振り分けアルゴリズム 重み付きラウンドロビン のサービスを定義.
ipvsadm -A -t 10.0.0.1:80 -s wrr
# 定義したサービスに対して パケット転送方式 NAT, 重み 1 の振り分け先を定義.
ipvsadm -a -t 10.0.0.1:80 -r 192.168.0.1:80 -m -w 1
# 振り分け先を削除
ipvsadm -d -t 10.0.0.1:80 -r 192.168.0.1:80
# サービスを削除
ipvsadm -D -t 10.0.0.1:80
```

図 22　ipvsadm によるベンチマークプログラム

```
#!/usr/local/src/mruby/bin/mruby
# 仮想 IP アドレス 10.0.0.1 ポート 80 番
# 振り分けアルゴリズム 重み付きラウンドロビン のサービスを定義
svc = IPVS::Service.new({'addr' => '10.0.0.1',
                         'port' => 80, 'sched_name' => 'wrr'})
# IP アドレス 192.168.0.1 ポート 80 番 重み 1 の振り分け先を定義
dest = IPVS::Dest.new({'addr' => '192.168.0.1',
                       'port' => 80, 'weight' => 1})
# サービスの追加
svc.add_service
# 振り分け先の追加
svc.add_dest dest
# 振り分け先の削除
svc.del_dest dest
# サービスの削除
svc.del_service
```

図 23　本システムによるベンチマークプログラム

```
#!/bin/bash
# 1000 回 引数に与えられたプログラムを繰り返し実行する
bench_prog=$1
for i in `seq 1 1000`; do
  ${bench_prog}
done
```

図 24　ベンチマーク用シェルスクリプト

ポート番号: 80
振り分け方式: NAT
重み: 1

ベンチマークに用いたプログラムを図 22，図 23 に示す．図 24 のようなシェルスクリプトを用いて，bench_prog 変数をそれぞれのプログラムに置き換え，Linux コマンドの time コマンドを用いて計測した．

結果を表 6.2 に示す．ipvsadm より高速に動作しているため，実運用に耐えられる速度であ

ることがわかる.

表 1　ベンチマーク結果

	本システムを用いた操作	ipvsadm を用いた操作
real	4.711s	5.569s
user	2.273s	0.764s
sys	1.914s	1.730s

7　関連研究

ldirectord [7] は，ipvsadm コマンドを，スクリプト言語である perl でラップしたものである．Keepalived と同じく，独自の DSL を用いて IPVS の設定を行う．ldirectord の DSL も，Keepalived と同じく制御構文が使えないことや，ヘルスチェックの方法が限定的であるなどの問題点が存在する．本研究では，プログラミング言語である mruby 上から IPVS を操作できる機構を提案した．

mod_mruby [8] は， Web サーバソフトウェアである Apache の機能拡充のための mruby 用インタフェースを実装したものである．mod_mruby は， mruby スクリプト実行時に生成される領域やライブラリのロードを，複数の mruby スクリプトで共有し，コンパイル済みのバイトコードのみを使い分けることにより，高速に動作するように設計されている．本研究では，Linux ロードバランサである IPVS を mruby から操作できるインタフェースを実装した．

8 おわりに

8.1 本研究のまとめ

本論文では，Linux のロードバランサである IPVS に対して，組み込み向け Ruby である mruby によるインタフェースを実装した．

実装したインタフェースを用いて，簡単な振り分け設定を記述することにより，既存のインタフェースである Keepalived ならびに ipvsadm と比較した．mruby の高い記述性を活かし，簡単に DSL のような構文が実装可能なことや，制御構文を用いた動的な振り分け先の操作が可能であること，また，外部ライブラリを用いて必要に応じて機能を拡張可能であることを示した．

また，実装したインタフェースに対し，ベンチマークを行うことで，ipvsadm より高速に動作しており，実運用に耐えられる速度であることを示した．

8.2 今後の課題

本論文で提案したインタフェースには, libipvs の全ての機能は実装済みでないため, libipvs の全ての関数を mruby から呼び出せるように実装を進めることが必要である．更に，mod_mruby [8] のように，高速化が期待できる箇所も複数あると考えられる．

本論文で提案したものはあくまでインタフェースであり，より高度な IPVS の操作や，ロードバランサとしての機能を満たすための機能拡充の余地は大きい．そのため，mruby-ipvs を用いて，mruby で大規模なソフトウェアを記述したり，更なる機能を求め，新たな mrbgem などを実装し，組み合わせることで，よりよいものが提案可能である．

謝辞

本研究を行うにあたり，終始変わらぬ御指導を賜りました中野圭介准教授，岩崎英哉教授，鵜川始陽助教に深く感謝致します.

参考文献

[1] The linux virtual server project. http://www.linuxvirtualserver.org/.

[2] Keepalived for linux. http://www.keepalived.org/.

[3] mruby. https://github.com/mruby/mruby.

[4] mruby-curl. https://github.com/mattn/mruby-curl.

[5] mruby-process. https://github.com/iij/mruby-process.

[6] mruby-sleep. https://github.com/mattn/mruby-sleep.

[7] ldirectord. http://horms.net/projects/ldirectord/.

[8] 亮介 松本 and 寿男 岡部. mod_mruby: スクリプト言語で高速かつ省メモリに拡張可能な web サーバの機能拡張支援機構. In **インターネットと運用技術シンポジウム** *2013* **論文集**, volume 2013, pages 79–86, dec 2013.