筑波大学大学院博士課程

システム情報工学研究科修士論文

分散システムのためのメッセージ表現手法に関する研究

古橋 貞之

(コンピュータサイエンス専攻)

指導教員　新城靖

2012 年 3 月

概要

今日では、プログラムを同時並行的に複数のサーバ上で実行し、ネットワークを介して互いにメッセージを交換しあうことで構成される分散システムが、幅広く利用されている。それらのシステムは、スクリプト言語を含む様々なプログラミング言語で実装されたプログラムが、新旧入り混じった様々なサーバ上で実行され、ソフトウェアおよびハードウェアのアップデートを繰り返しながら長期間運用される。分散システムは、このような異種混合の環境においても相互運用性を維持し、動作し続けなければならない。

そこで本研究では、分散システムのための新たなメッセージ表現手法を設計し実装した。これを MessagePack と命名する。MessagePack は、特定のプログラミング言語に依存しない中立的な型システムと、その型とプログラミング言語の型との相互変換を行う機構を導入することによって、異種のサーバ・異種のプログラミング言語・異種のバージョンのプログラムの間で互換性を維持しながら、メッセージを交換することを可能にした。さらに、コンパクトに型情報を格納するバイナリ形式の表現形式を設計することによって、優れた空間効率を達成した。また、実装においても最適化を行うことによって、既存のオブジェクトシリアライズ手法と比較して、優れた処理速度を実現した。

本稿では MessagePack の設計と実装について述べ、その評価を行う。

目 次

図目次

第1章　はじめに

今日では、多種多様なサービスをネットワークを通じて多数のユーザーに提供するシステムの構築方法が一般的となっている。これらのシステムを実装するには、次に挙げる 3 つの特徴を考慮する必要がある：

第 1 に、それらのサービスは多数の機能を組み合わせて構築される。例えば、写真を Web 上で管理できるようにするサービスは、写真を閲覧する機能や、写真を保存しておく機能、写真を検索する機能などを組み合わせて構築される。このとき、個々の機能はそれを実装するのに適したプログラミング言語で実装されている場合がほとんどである [1]。例えば、Web アプリケーションの実装には Ruby、データベースの実装には C++、検索エンジンの実装には Java というように、様々なプログラミング言語を組み合わせて一つのシステムが実装される。これらの多種のプログラミング言語を横断してメッセージを交換しあう必要がある。

第 2 に、それらのサービスは多くのユーザーから同時にアクセスされる可能性があるため、高い負荷に耐えられるように実装しなければならない。また、その負荷が将来的にどのように増大していくかを事前に予測することは難しい。このようなシステムを経済的に構築するには、1 台の計算機をより高性能なものに置き換えていくことで性能向上を図る Scale-up 型の手法より、安価な計算機を連携させて分散システムを構成する Scale-out 型の手法の方が有効であることが知られている [2]。しかし、計算機の台数が多くなるほど通信のオーバーヘッドが増大していくため、性能の劣化を防ぐ最適化が必要になる。また、システムを構成する計算機の一部が故障する確率が増大するため、部分的な故障が全体的な故障に繋がらないようにしなければならない。このため、Scale-out 型の分散システムは実装の難易度が高い。

第 3 に、それらのサービスはソフトウェアやハードウェアの更新を繰り返しながら長期間運用されることがある。結果として、単一のサービスを利用するクライアントおよびサービスを提供するシステムの中で、バージョンが最新でないプログラム、性能が均質でないサーバ、あるいは CPU アーキテクチャの異なる端末などが混在する。実装及び運用のコストを低減しながらも、それらの間で相互運用性を維持し続けなければならない。

以上の特徴から、異種のプログラミング言語・異種の計算機環境・異種のバージョン間で互換性を維持しながら利用可能であり、高い相互運用性を実現する、効率の良いメッセージ表現手法が必要とされている。実装の難易度を低減するために、そのソフトウェアライブラリとしての使い勝手も重要になる。

本研究の目的は、以上のような背景を踏まえ、多種のプログラミング言語を横断して利用することができ、異種混合の計算機環境においても高い相互互換性を実現し、高速に動作するメッセージ表現手法を設計および実装することで、分散システムの開発を容易にすること

である。新たなメッセージ表現手法を設計し、実装を行うことによって、この目的を達成する。このメッセージ表現手法を **MessagePack** と命名する [3]。

MessagePack では、特定のプログラミング言語に依存しない中立的な型システムである「MessagePack の型システム」と、その型システムをバイナリ列に変換して表現する「MessagePack の表現形式」を導入することによって、多種のプログラミング言語を横断してメッセージを交換することを可能にする。MessagePack の表現形式では、コンパクトに型情報を格納する手法を導入することによって、優れた空間効率を実現する。また、MessagePack の型システムとプログラミング言語の型との相互変換を行う枠組み「型変換テンプレート」を導入し、その拡張を可能にすることによって、プログラミング言語処理系やライブラリによって提供される複雑な型のシリアライズをも容易にする。さらに、実装においても最適化を行い、優れた処理速度を実現する。

本論文の構成は次のようになっている。第 2 章では、本研究に関連した技術について述べる。第 3 章では、MessagePack の設計について述べる。第 4 章では、MessagePack のプログラミング言語に対するバインディングについて述べる。第 5 章では、MessagePack の実装と最適化について述べる。第 6 章では、MessagePack の評価について述べる。最後に 7 章では、本論文のまとめを行う。

第2章　関連研究

本章では、本研究に関係する既存の研究や技術について述べる。

2.1　XDR

XDR[4] は、オブジェクトシリアライズ手法の一つである。バイナリ形式の表現方法を用いるため、空間効率および処理速度に優れる。

XDR ではシリアライズ対象となるオブジェクトの構造を記述するために、専用のインタフェース記述言語（IDL）を用いる。この記述から、C の構造体の定義と、その構造体をシリアライズ・デシリアライズするプログラムが生成される。この IDL の記述例を次に示す。

```
struct echo_message {
    string str;
    int num;
};
```

struct echo_message は、文字列と整数を含んだ構造体の定義である。

XDR では、シリアライズされたデータの中に元の構造体の型に関する情報は含まれていないため、IDL が無ければオブジェクトをデシリアライズすることができない。また、XDR の処理系である rpcgen は C のプログラムを生成することから、特別な工夫なしには異種のプログラミング言語間でデータを交換することはできない。

これに対して本研究では、データの中に型情報を格納することで、IDL 無しでオブジェクトをデシリアライズすることを可能にする。また、多言語間でデータを交換することを可能にする。

2.2　SunRPC

SunRPC[5] は、RPC(Remote Procedure Call) システムの 1 つである。SunRPC はシリアライズの方式として XDR を用いる。

SunRPC では、XDR による構造体の定義と共に、RPC のインタフェースを記述する。文字列と整数を含んだ構造体を送受信するプログラムを記述する例を次に示す。

```
struct echo_message {
    string str;
    int num;
};
```

```
program ECHO_PROG {
    version ECHO_VERSION {
        echo_message echo(echo_message) = 1;
    } = 1;
} = 0x20001001;
```

programセクションの中にRPCのインタフェースを記述している。echo_message echo(echo_message) = 1; では、RPC によって呼び出すことができる関数のインタフェースを定義している。

SunRPC は、やりとりするデータの型に関する情報を交換する方法を提供しない。このため、サーバ・クライアントの双方が同じインタフェース定義を事前に共有しておかなければならず、同じインタフェース定義を共有していなければ、データを交換することができない。

本研究では、動的型付けオブジェクトを利用することによって、インタフェース定義を事前に共有することなくデータを交換することを可能にする。

プログラムの更新と共にインタフェースの更新が必要になることは少なくないが、SunRPC ではインタフェース定義が更新されることを想定しており、RPC のメッセージの中にバージョン番号を埋め込むことができる。クライアントとサーバのインタフェース定義のバージョンが異なれば、RPC は正しく失敗する。

一方本研究では、交換されるオブジェクトの型情報を利用したプログラミングを可能にすることによって、異なるバージョンのプログラムでも正しくメッセージを交換できるようにする。

2.3 JSON

JSON[4] は、オブジェクトシリアライズ手法の一つである。テキスト形式を用いるため、空間効率および処理速度は劣る。

JSON は XDR とは異なり、IDL を必要としない。オブジェクトをシリアライズする際に、型に関する情報を同時に格納することで、IDL を参照せずにオブジェクトをデシリアライズすることを可能にしている。

また、JSON は数多くの言語で実装されており、異種の言語間でオブジェクトを交換するために使用することができる。

一方、IDL を使用しないことから、デシリアライズされたオブジェクトの型は実行時に初めて決定することになるが、C++や Java などの静的型付け言語では、そのような動的型付けオブジェクトは扱いにくい。このため、デシリアライズされたオブジェクトを静的な型に変換するために、冗長なプログラミングが多く必要になるという欠点がある。

本研究では、動的型付けオブジェクトを静的に宣言された型に変換する API を実装することによって、静的型付け言語においても安全かつ高速にデシリアライズされたオブジェクトを扱えるようにする。

2.4 Thrift

Thrift[6] は、RPC システムの一つである。バイナリ形式を用いるため、空間効率および処理速度に優れる。大規模 SNS(Social Network Service) サービス事業者の Facebook によって開発され、Facebook で利用されている。

Thrift を利用するには、SunRPC と同様に、専用の IDL を使って RPC のインタフェースを記述しなければならない。この IDL を使用して、文字列と整数を含んだメッセージを送受信するインタフェースを定義する例を次に示す。

```
struct EchoMessage {
    2:string str
    1:i32 num,
}

service EchoServer {
    EchoMessage echo(1: EchoMessage msg)
}
```

struct EchoMessage は、送受信するメッセージを表すデータ構造を定義している。EchoMessage echo(1: EchoMessage msg) では、RPC を使って呼び出すことができる関数のインタフェースを定義している。

Thrift は SunRPC とは異なり、メッセージの中にバージョン番号を埋め込むことができない。このためクライアントとサーバで異なるバージョンの IDL を使用していた場合に、それが必ずしも正しく検出されるとは限らない。このため転送されたデータが誤った形でデシリアライズしてしまう恐れがある。

この問題に対して、Thrift では安全に IDL を更新できる方法を制限している。すなわち、構造体のフィールドに付与する一意な数値を更新してはならない。Thrift を利用する開発者がこの制限に従っている限りは、正しくオブジェクトを交換することができる。

本研究では、IDL を使用して型を静的に解決する代わりに、交換されるオブジェクトの型情報を利用したプログラミングを可能にすることによって、異なるバージョンのプログラムでも正しくメッセージを交換できるようにする。

2.5 Protocol Buffers

Protocol Buffers は、RPC システムの一つである [7]。オブジェクトシリアライズ手法にはバイナリ形式を用いるため、空間効率および処理速度に優れる。Google が開発し、大規模なシステムで実際的に利用されている。Protocol Buffers はいくつかのプログラミング言語で実装されており、異種の言語をまたいでオブジェクトをやりとりすることができる。

Protocol Buffers を利用するには、専用の IDL を使って定義を記述しなければならない。この IDL を使って、文字列と整数を含んだメッセージを記述する例を次に示す。

```
package proto_protobuf;
option optimize_for = SPEED;
```

```
message EchoMessage {
    required string str = 1;
    required  int32 num = 2;
}
```

message EchoMessage は、RPC によって呼び出すことができる関数のインタフェースを定義している。この関数は str と num という 2 つの引数を受け取る。str の型は文字列であり、num の型は 32 ビットの符号付き整数である。

本研究では、IDL を使用せずに異種のプログラミング言語間・異種のサーバ間・異種のバージョン間でメッセージを交換することを可能にする。

2.6 Avro

Avro[8] は、RPC システムの一つである。オブジェクトシリアライズ手法にはバイナリ形式を用いる。Avro は XDR や Thrit などと同様に IDL を記述しなければならないが、交換するデータの先頭に IDL を付与することで、必ずしも事前に共有する必要が無いようにしている。

XDR や Protocol Buffers では、事前に共有されている IDL からプログラムを生成する方法を採っているため、IDL を解析・評価する処理系は 1 度実装すれば多数の言語に対応できる。しかし Avro では、交換されるデータの先頭に IDL が付与されている場合に対応するために、すべての言語で IDL の処理系を実装しなければならない。このため多言語への実装の移植が難しいという欠点がある。Avro はこの実装コストの問題をある程度軽減するために、IDL の表現に JSON を用いているが、JSON はテキスト形式であり効率は劣る。

Avro は異種の言語をまたいで利用することを想定したシステムであるが、現時点で利用できる言語は C、Java および Python の 3 種類のみである。

本研究では、最低限利用可能な仕様をシンプルにすることによって実装を簡略化し、より多くの言語で利用できるようにする。同時に、発展的な機能として動的に型付けられたオブジェクトを静的に宣言された型に変換する API を提供することにより、IDL を使用とせずとも高速にメッセージを扱えるようにする。

2.7 JavaRMI

Java RMI(Remote Method Invocation)[9] は、ネットワークをまたいでオブジェクトのメソッド呼び出しができるようにすることを目的とした、Java のライブラリである。

Java RMI は、Java でしか利用できない。本研究では、複数のプログラミング言語を横断してオブジェクトを交換できるようにする。

2.8 CORBA

CORBA(Common Object Request Broker Architecture) は、OMG(Object Management Group) によって仕様が定められている分散オブジェクト技術である。CORBA は、異なるプログラ

ミング言語で構成された分散環境で動作するシステムを実装するための機能を提供している。しかし、API が必要以上に複雑であるという問題が指摘されている [10]。

本研究では、実装を容易にして多言語対応を容易にするために、その設計をシンプルに抑えた。また、複雑な機能は必ずしも実装しなくても利用できるようにし、段階的な拡張が可能な設計とした。

2.9 既存のメッセージ交換手法の課題

既存のメッセージ交換手法には、次のような課題がある：

異種のプログラミング言語をまたいで利用できない XDR、SunRPC、JavaRMI

静的型検査の仕組みを持たない JSON

IDL を事前に共有しておかなければならない Thrift、Protocol Buffers

複雑で実装が難しい CORBA、Avro

空間効率と処理速度が低い JSON、Avro

MessagePack では、これらの課題を解決する。

第3章　MessagePackのモデルと設計

本章では、本研究で実装したシリアライズ形式であるMessagePackの設計について述べる。第1節では、MessagePackで解決するメッセージ交換手法の課題を述べる。第2節では、課題を解決するためモデルについて述べる。第3節では、MessagePackの型システムについて述べる。最後に第4節では、MessagePack表現形式について述べる。

3.1　メッセージ交換手法の課題

まず、異種の言語をまたいでメッセージを交換することを可能にするには、それぞれの言語で提供されている型システムを相互に変換しなければならない。例えば、C++では文字列とバイト列は区別されないが、JavaやRubyでは区別される。同じRubyにおいても、バージョン1.8にでは文字列とバイト列は区別されなかった。また、Javaでは多倍長整数型や日付型が標準ライブラリとして提供されるが、C++では提供されない。このように、プログラミング言語によって型システムはまったく異なるため、あらゆる言語に完全に適合する型システムを設計することは難しい。

また、シリアライズされたデータからオブジェクトをデシリアライズして復元するには、元のオブジェクトの型に関する情報が必要になる。この型情報を、シリアライズ結果のデータの中に埋め込む方法を採用すると、データのサイズが肥大化してしまう問題がある。一方で、別の場所に格納する方法、すなわちIDL(インタフェース定義言語)を使用する方法を採用すると、離れたプログラムの間でIDLを事前に共有しておく方法を用意しなければならない。ソフトウェアが日々更新されていく中で、データとIDLのバージョンを常に一致させておくことは難しい。

3.2　MessagePackの型変換モデル

前節で述べた問題とトレードオフを解決するために、MessagePackは次のような設計とした：

- どのプログラミング言語にも依存しない、中立的な型システムを定義し、この型システムとプログラミング言語の型システムを相互に変換する仕組みを実装することで、多数のプログラミング言語を横断してメッセージを交換することを可能にする
- 型情報をシリアライズ後のデータの中に格納するが、ハフマン符号化[11]に見られるように、想定される使用頻度に応じてビットを割り当てることにより、型に関する情報

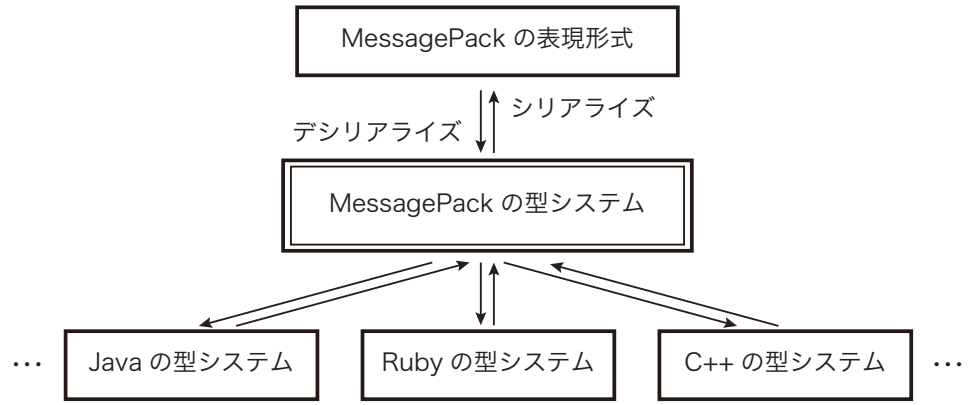

図 3.1: MessagePack の型変換モデル

をコンパクトに格納することを可能にする

MessagePack では、どのプログラミング言語にも依存しないデータ型の集合を扱う。この中立的なデータ型の集合を **MessagePack の型システム**と呼ぶ。また、それらのデータ型をバイナリ列に変換して表現できるようにする。この MessagePack の型システムのバイナリ列における表現形式を **MessagePack の表現形式**と呼ぶ。MessagePack は、中立的な MessagePack の型システムと、バイナリ形式での表現形式および各言語の型システムを相互に変換することによって、多言語間のメッセージ交換を可能にする。このモデルを図 3.1 に示す。

3.3 型システムの設計

JSON は RFC 4627[12] で標準化されたシリアライズ形式であり、Web システムを中心に幅広く利用されている。JSON の型システムは多くの開発者が習熟しているものである。そこで MessagePack の型システムで扱うデータ型は、JSON で扱うデータ型とほぼ同じになるようにした。これによって、新たな型システムを習熟しなければならないコストを低減する。

MessagePack の型システムで扱う 7 種類のデータ型を次に示す。

Nil Nil は、値がないことを表す。

Boolean Boolean は、真偽値を示す型である。True または False を表す。

Integer Integer は、整数を示す型である。64 ビットの符号付き整数で表せる最小値（つまり-9,223,372,036,854,775,808）から、64 ビットの符号無し整数で表せる最大値（つまり 1,844,674,407,370,955,1615）までの整数を表す。

Float Float は、浮動小数を表す型である。IEEE 754[13] で定義された単精度二進化浮動小数点数または倍精度二進化浮動小数点数で表せる数を表す。

Raw Raw は、バイト列を表す型である。

Array Array は、配列を表す型である。

Map Map は、連想配列を表す型である。

これらのデータ型は、ほとんどの言語で利用することができるプリミティブなデータ型である。データ型の種類を少なくすることにより、実装の難易度が下がるため、移植性が向上し、結果として多言語対応が容易になるという利点が得られる。

プログラミング言語によっては日付型や多倍長整数型のような型を提供しているが、そのような型は上記の MessagePack 型に射影してシリアライズを行う。このような発展的な型に対する射影関数の導入については、4 章で後述する。

3.4 表現形式の設計

3.4.1 型情報の格納

MessagePack の表現形式では、シリアライズ後のデータの中に、オブジェクトの型を表す情報を格納する。型情報を格納することで、事前に IDL を共有しておくことなくオブジェクトを交換することが可能になる。

また、データの中にオブジェクトの型を表す情報が格納されていると、プログラムが想定していない型のデータを受け取った場合に、それを検出することができる。言い換えれば、データに冗長性を持たせることができる。これによって、誤って送られてきたデータや、破損したデータを検出することが可能になる。

さらに、プログラムが想定していないデータを受け取った場合でも、型からデータの意味をある程度推測することができる。このようにデータの意味を推測できると、間違ったデータを送信してしまうようなバグがあった場合に、どの部分のプログラムが原因で問題が起こっているかを突き止めるための手がかりが増える。これによって、メッセージを交換するプログラムのデバッグを容易にしている。例えば、ネットワークアナライザである wireshark[14] を用いてパケットをスニッフィングし、得られたデータをデシリアライズして表示することが可能になった [15]。実際に運用しているソフトウェアを改変したり再起動したりすることなく、やりとりされているデータの構造を可視化し、デバッグに役立てることができる。

3.4.2 データサイズの削減

前節で述べた利点の一方で、型情報を含めるとデータのサイズは増大してしまう。この問題を軽減するために、型情報をコンパクトに格納できるように表現形式を設計した。ハフマン符号 [11] に見られるように、より多く出現すると想定される種類の値に対して、より多くのビットを割り当てる。これによってデータサイズを削減する。

MessagePack の表現形式では、先頭の 1 バイトの中でデータの型を表し、そのバイトまたはそれ以降のバイトで値を表す。先頭の 1 バイトは、次に示すように複数のグループに分類する。

表現形式	2進数表現	16進数表現
Fixnum		
Positive Fixnum	0xxxxxxx	00 - 7f
Negative Fixnum	111xxxxx	e0 - ff
Category-1		
nil	11000000	c0
(reserved)	11000001	c1
false	11000010	c2
true	11000011	c3
Category-2		
float	11001010	ca
double	11001011	cb
uint 8	11001100	cc
uint 16	11001101	cd
uint 32	11001110	ce
uint 64	11001111	cf
int 8	11010000	d0
int 16	11010001	d1
int 32	11010010	d2
int 64	11010011	d3
Category-3		
raw 16	11011010	da
raw 32	11011011	db
array 16	11011100	dc
array 32	11011101	dd
map 16	11011110	de
map 32	11011111	df
FixRaw		
FixRaw	101xxxxx	a0 - bf
FixArray		
FixArray	1001xxxx	90 - 9f
FixMap		
FixMap	1000xxxx	80 - 8f

それぞれのグループには、次のような規則性を設定した。

Fixnum このグループに属する表現形式は、オブジェクトの型と値が 1 バイトで表される。また、その 1 バイトがそのまま（C ではキャストするだけで）符号無し 8 ビット整数または符号付き 8 ビット整数の値になる。

Category-1 このグループに属する表現形式は、オブジェクトの型と値が 1 バイトで表される。

Category-2 このグループに属する表現形式は、オブジェクトの型と、後続するデータの長さが、1 バイトで表される。 また、そのデータの長さは (1 << (先頭バイト & 0x03)) という計算式によって求められる。

Category-3 このグループに属する表現形式は、オブジェクトの型と、後続するオブジェクトの要素数を表すデータの長さが、1 バイトで表される。また、そのオブジェクトの要素数を表すデータの長さは (2 << (先頭バイト & 0x01)) という計算式によって求められる。

一つのオブジェクトを表すために複数の表現形式を選択できる場合は、もっとも小さいサイズで表現できる表現形式を選択する。例えば、整数 1 や整数 2 は、表現形式 Positive Fixnum を使用して 1 バイトで表現する。このように、フラグやエラーコードなどの用途に頻繁に利用される小さい整数は、このようにごく少ないバイト数で表現することができる。同様に、エラーメッセージの表現などで頻繁に利用される短い文字列や、構造体の表現に頻繁に使用される短い配列も、少ないバイト数で表現することができる。一方、整数 70000 は、表現形式 uint32 を使用して 5 バイトで表現する。このように、使用される頻度が少ない大きなサイズのオブジェクトは、多くのバイト数を消費して表現する。

表現形式の一覧は付録に示す。

第4章　プログラミング言語バインディング

３章では、「MessagePack の型システム」と、その型システムのバイナリ列によって表現する「MessagePack の表現形式」について述べた。この章では、様々なプログラミング言語へのバインディングについて述べる。その特徴は、第１に、静的な型付けを採用するプログラミング言語においても、MessagePack の型システムに基づいた動的な型付けを利用可能にした点にある。第２の特徴は、型変換テンプレートと呼ぶ機構を導入することによって、MessagePack の型システムと各言語の型システムの相互変換を可能にした点にある。

4.1　動的型付けオブジェクト API

3 章で述べたように、MessagePack ではシリアライズされたバイト列の中に型の情報を含める。この型は必ずしも静的には決定せず、デシリアライズを行う実行時になって初めて決定する。

このような動的型付けオブジェクトを使用すると、オブジェクトの型を利用した柔軟なプログラミングが可能になる。例えば、受け取ったデータの型が Raw 型である場合と Integer 型である場合で異なる処理を行うようなプログラムを記述できる。このようなプログラミング手法は、他のソフトウェアと通信の相互互換性を保ったままプログラムを更新していく必要がある分散システムにおいて特に有効である。

MessagePack では、この動的型付けオブジェクトの特性を利用できるようにするために、MessagePack の型システムを実装した API をライブラリとして提供した。これによって、静的な型付けを採用するプログラミング言語においても、MessagePack の型システムに基づいた動的な型付けを利用することが可能になる。この API を動的型付けオブジェクト API と呼ぶ。

C++で、動的型付けオブジェクト API を使用し、デシリアライズしたオブジェクトの型によって処理を分岐するプログラムを次に示す。

```
#incldue <iostream>
#include <memory>
#include <msgpack.hpp>

void use_integer(msgpack::object obj) {
    // short型に変換して扱う
    short n = obj.as<short>();
}

void process(msgpack::object obj, std::auto_ptr<msgpack::zone> z)
try {
    if(obj.type == msgpack::type::ARRAY) {
        // 実際の型が配列なら配列の要素に対してuse_integer()を複数回呼び出していく
```

```
        msgpack::object_array array = obj.via.array;
        for(int i=0; i < array.size; i++) {
            use_integer(array.ptr[i]);
        }
    } else {
        // そうでなければuse_integer()を1回呼び出す
        use_integer(obj);
    }
} catch (msgpack::type_error&) {
    // 型変換に失敗した場合は例外が発生する
    std::out << "expected integer or array of integer: " << std::endl;
}

int main(void) {
    msgpack::unpacker unpacker;

    while(true) {
        // 標準入力からデータを読み出す
        unpacker.reserve_buffer(32*1024);
        size_t len = std::in.readsome(
            unpacker.buffer(), unpacker.buffer_capacity());
        unpacker.buffer_consumed(len);

        // デシリアライズを行って動的型付けオブジェクトを得る
        while(unpacker.execute()) {
            std::auto_ptr<msgpack::zone> z(unpacker.release_zone());
            msgpack::object obj = unpacker.data();  // 動的型付けオブジェクト
            unpacker.reset();

            process(obj, z);
        }
    }
}
```

同様の処理を Java で実装したプログラムを次に示す。

```
import org.msgpack.MessagePack;
import org.msgpack.unpacker.Unpacker;
import org.msgpack.type.Value;
import org.msgpack.type.IntegerValue;

public class MyReader {
    public static void main(String[] args) throws Exception {
        // デシリアライズを行って動的型付けオブジェクトを得る
        Unpacker unpacker = new MessagePack().createUnpacker(System.in);
        Value obj = unpacker.read();  // 動的型付けオブジェクト

        try {
            if(obj.isArrayValue()) {
                // 実際の型が配列なら配列の要素に対して
                // useInteger()を複数回呼び出していく
                for(Value v : obj.asArrayValue()) {
                    useInteger(v.asIntegerValue());
                }
            } else {
                // そうでなければuseInteger()を1回呼び出す
                useInteger(obj.asIntegerValue());
            }
        } catch (MessageTypeException ex) {
            // 型変換に失敗した場合は例外が発生する
            System.out.println("expected integer or array of integer: "+ex);
        }
```

```
    }

    // IntegerValueを使用して処理を行う
    public void useInteger(IntegerValue iv) {
        // short型に変換して扱う
        Number num = iv;
        short s = num.shortValue();
        System.out.println(s);
    }
}
```

動的型付けオブジェクトには、静的に宣言された型に変換するメソッドを実装している。この変換が失敗した場合、すなわち想定していた型と実際に動的に決定された型が異なっていた場合は、例外を発生させる。

また MessagePack の Java 実装では、動的型付けオブジェクトは equals や hashCode を正しく実装している。このため静的に宣言された型に変換することなく、同一性を判定したり HashMap のキーとして利用したりすることができる。この例を次に示す。

```
import java.util.HashSet;
import java.util.ArrayList;
import org.msgpack.MessagePack;
import org.msgpack.unpacker.Unpacker;
import org.msgpack.type.Value;
import org.msgpack.type.IntegerValue;

public class MyReader {
    public static void main(String[] args) throws Exception {
        HashSet<Value> unique = new HashSet<Value>();
        ArrayList<Value> all = new ArrayList<Value>();

        // ストリームデシリアライザを使用して、MessagePackストリームから
        // オブジェクトを次々にデシリアライズする
        Unpacker unpacker = new MessagePack().createUnpacker(System.in);
        for(Value obj : unpacker) {
            // HashSetにaddすることができる
            // 同一性判定はMessagePackの型システムに基づいて行われる
            unique.add(obj);
            all.add(obj);
        }

        int duplicated = all.size() - unique.size();
        System.out.println("num of duplicated values: "+duplicated);
    }
}
```

4.2 型変換テンプレート

前節で述べた利点がある一方で、C++や Java などの静的型付け言語では、動的に型付けられたオブジェクトを簡単に扱う仕組みがないため、動的型付けオブジェクトは扱いにくい。

そこで MessagePack では、デシリアライズされた動的型付けオブジェクトを静的に宣言された型に変換する API をライブラリとして提供した。これによって MessagePack の利用者は、

型検査を行うプログラムを自分で記述することなく、デシリアライズされたオブジェクトを静的な型に変換して安全に扱うことが可能になる。

この機能は、プログラミング言語の型システムから MessagePack の型システムへ（シリアライズ時)、または MessagePack の型システムからプログラミング言語への型システムへ（デシリアライズ時）への射影関数を導入することを意味する。MessagePack では、この型変換を行う射影関数の実装を型変換テンプレートと呼ぶ。

Java で、型変換テンプレートを用いて動的型付けオブジェクトを静的に宣言された型に変換する例を示す。

```
import org.msgpack.MessagePack;
import org.msgpack.unpacker.Unpacker;
import org.msgpack.type.Value;
import org.msgpack.type.IntegerValue;

public class MyReader {
    static MessagePack msgpack = new MessagePack();

    public static void main(String[] args) throws Exception {
        // デシリアライズを行う
        Unpacker unpacker = msgpack.createUnpacker(System.in);
        Value obj = unpacker.read();

        // 動的型付けオブジェクトからint型の配列へと型変換を行う
        int[] converted = msgpack.convert(obj, int[].class);
}
```

4.2.1 型変換テンプレートの拡張

MessagePack では、型変換テンプレートを利用者が拡張できるようにした。これによって、ユーザーが独自に定義したクラスに対する射影関数を導入することが可能になる。MessagePack の利用者は、MessagePack が提供する枠組みを使用して型変換を行う処理を記述すれば、プログラミング言語で標準的に提供されている他の型とまったく同じように独自クラスのインスタンスをシリアライズ・デシリアライズすることが可能になる。

また同時に、型変換テンプレートを拡張可能なように実装しておくことで、プログラミング言語が更新された際に追加された型に対する射影関数を導入することが容易になるという利点も得らる。

Java で、型変換テンプレートを拡張することにより、多倍長整数を扱えるようにする例を示す。

```
import java.io.IOException;
import java.math.BigInteger;
import org.msgpack.MessagePack;
import org.msgpack.packer.Packer;
import org.msgpack.packer.BufferPacker;
import org.msgpack.unpacker.Unpacker;
import org.msgpack.unpacker.BufferUnpacker;
import org.msgpack.template.AbstractTemplate;
import org.msgpack.MessageTypeException;
import org.msgpack.type.Value;
```

```
public class MyModel {
    static MessagePack msgpack = new MessagePack();

    // 型変換テンプレートを定義
    public static class BigIntegerTemplate extends AbstractTemplate<BigInteger> {
        public BigIntegerTemplate() {}

        public void write(Packer pk, BigInteger target, boolean required)
                throws IOException {
            if (target == null) {
                if (required) {
                    throw new MessageTypeException("Attempted to write null");
                }
                pk.writeNil();
                return;
            }
            pk.write((BigInteger) target);
        }

        public BigInteger read(Unpacker u, BigInteger to, boolean required)
                throws IOException {
            if (!required && u.trySkipNil()) {
                return null;
            }
            return u.readBigInteger();
        }
    }

    static {
        // 型変換テンプレートを登録する
        msgpack.register(BigInteger.class, new BigIntegerTemplate());
    }

    public static void main(String[] args) throws Exception {
        // デシリアライズを行う
        Unpacker unpacker = msgpack.createUnpacker(System.in);
        Value obj = unpacker.read();

        // 動的型付けオブジェクトからBigIntegerへの型変換が可能になる
        BigInteger converted = msgpack.convert(obj, BigInteger.class);
    }
}
```

型変換テンプレートを拡張することで、MessagePack によって標準的に提供されていない型でも型変換が可能になる。また、その使い勝手は型変換テンプレートの拡張を用いない前節の型変換と同じである。

4.2.2 実装の難易度と実装の段階的な拡張

以上のように拡張な型変換テンプレートを実装することで API の使い勝手は向上するが、その実装は難しい。しかし、型変換テンプレートは実装されていなくても MessagePack の利用は可能である。この場合、利用者は動的に型付けられたオブジェクトを直接利用することになる。特に動的型付け言語においては、動的型付けオブジェクトを直接扱うことは容易である。言い換えれば、拡張可能な型変換テンプレートは後付け可能な機能である。

3 章で述べたように、MessagePack の型システムでは少数のデータ型のみを提供する。このため、MessagePack は機能が制限された最小限の実装は容易に開発できる一方で、後から段階的に拡張していくことで、使い勝手を向上させることができる。

第5章　MessagePackの実装

MessagePack の実装では、MessagePack の表現形式のデータ・ストリームを簡単に、かつ、効率よく扱う必要がある。本研究では、様々な言語でシリアライズとデシリアライズを簡単に記述できるライブラリを用意した。さらに高速にデータを扱えるようにするために、処理のオーバーヘッドが小さくなるように工夫した。この章では、Ruby、Java、および C,C++における実装について述べる。

5.1　実装上の課題

本研究では、MessagePack を C++、Ruby、および Java の 3 つの言語において実装した。この節では、高速な実装を行うために重要なモジュールと実装上の課題について述べる。

5.1.1　ストリームの取り扱い

MessagePack の表現形式では、オブジェクトを表すデータに先立って、データの長さを格納する。このため、オブジェクトとオブジェクトの境界を区別することができる。複数のオブジェクトを次々にシリアライズして繋げたものを、**MessagePack ストリーム**と呼ぶ。これを図 5.1 に示す。MessagePack ストリームは、プロセス間通信を行うためのパイプや TCP ソケット上で利用するプロトコルや、ログファイルのファイル構造など、複数のオブジェクトを続けて交換する必要がある場合に広く利用される。このため、MessagePack ストリームを簡単に扱うための仕組みを実装する必要がある。

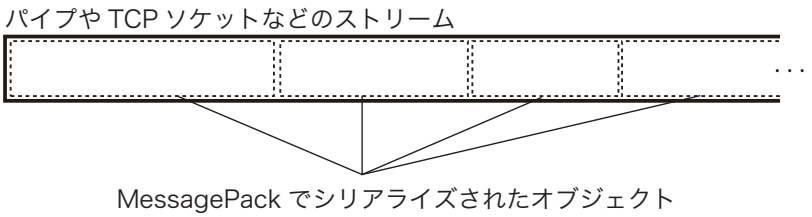

図 5.1: MessagePack ストリーム

5.1.2 コピーの削減

サイズの大きいメッセージを交換する場合、そのデータをコピーするオーバーヘッドは大きい。このため、このピーの回数を削減することが性能を向上させる上で重要になる。

5.1.3 メモリ確保の最適化

デシリアライザは、デシリアライズされたオブジェクトを保持するために多くのメモリを確保する。プログラミング言語処理系によっては、この負荷はバイナリデータの操作よりも大きなオーバーヘッドを生じる。従って MessagePack の実装では、オブジェクトの確保に伴うメモリの確保やガーベージコレクタ (GC) の負荷を軽減することが、性能を向上させる上で重要になる。

5.1.4 リフレクションの削減

MessagePack を実装するには、型に関する情報を扱うために、実行時にそれらの型情報を取り出す機能であるリフレクションが必要になることがある。しかし、リフレクションはオーバーヘッドが大きく、メモリおよび CPU 資源を多く消費する。従ってリフレクションの使用回数を抑えることが、性能を向上させる上で重要になる。

5.2 Ruby 実装の最適化

5.2.1 ストリームデシリアライザ

Ruby 実装では、MessagePack ストリームを扱う専用のクラスを用意した。これをストリームデシリアライザと呼ぶ。ストリームデシリアライザの利用方法を次に示す。

```
require 'msgpack'

# ストリームデシリアライザをインスタンス化
u = MessagePack::Unpacker.new

# 標準入力からデータを読み取る
while buf = STDIN.readpartial(1024)
  # ストリームデシリアライザにデータを供給
  u.feed(buf)

  # デシリアライズを進める
  u.each {|obj|
    # デシリアライズされたオブジェクトを表示
    puts "deserialized: #{obj}"
  }
end
```

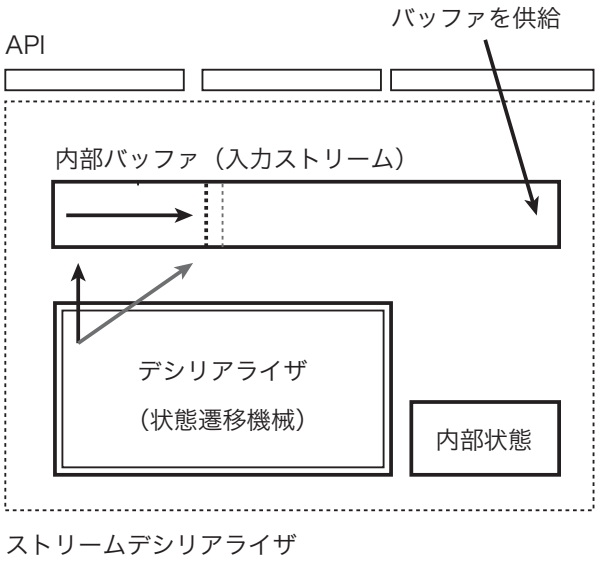

図 5.2: ストリームデシリアライザの構造

利用者は、パイプや TCP ソケットから読み取ったデータを、MessagePack::Unpacker#feed メソッドによってストリームデシリアライザに供給していく。そして MessagePack::Unpacker#each メソッドによってデシリアライズを進める。このように 2 つのメソッドを使用するだけで MessagePack ストリームを扱うことを可能とした。

このような利用方法を可能にするために、ストリームデシリアライザは内部にバッファを持つ。feed メソッドによって渡されたデータは、この内部バッファにコピーしていく。このとき、必要であれば realloc 関数を使用して領域を拡張する。また、ストリームデシリアライザには、MessagePack の表現形式を取り扱う有限状態遷移機械を実装した。この状態遷移機械は、内部バッファを入力データとして次々に読み取り、状態を遷移させていくことによってデシリアライズを行う。このように、MessagePack ストリームの取り扱い方法をクラスの内部に隠蔽することにより、簡便な API で MessagePack ストリームを扱えるようにした。ストリームデシリアライザの構造を図 5.2 に示す。

5.2.2 コピーの削減

feed メソッドはデータのコピーを行うため、効率が悪い。そこで、feed メソッドと each メソッドの機能を統合した feed_each メソッドを導入した。feed_each メソッドは、引数に与えられたデータが内部バッファに続いて繋がっていると見なして状態遷移機械にデータを入力し、デシリアライズを進める。feed_each メソッドの終了時に、状態遷移機械に入力されなかった端数のデータのみを内部バッファにコピーする。これによってコピーされるデータ量を削減した。利用方法を次に示す。

```
require 'msgpack'

# ストリームデシリアライザをインスタンス化
u = MessagePack::Unpacker.new

# 標準入力からデータを読み取る
while buf = STDIN.readpartial(1024)
  # デシリアライズを進め、余ったデータのみを内部バッファにコピーする
  u.feed_each(buf) {|obj|
    # デシリアライズされたオブジェクトを表示
    puts "deserialized: #{obj}"
  }
end
```

5.3 Java 実装の最適化

5.3.1 ストリームデシリアライザ

Java 実装では、Ruby 実装と同様にストリームデシリアライザを提供する。利用方法を次に示す。

```
import java.io.IOException
import org.msgpack.MessagePack;
import org.msgpack.type.Value;
import org.msgpack.unpacker.BufferUnpacker;
import org.msgpack.unpacker.UnpackerIterator;

public class IoEventHandler {
    // ストリームデシリアライザをインスタンス化
    private BufferUnpacker unpacker = new MessagePack().createBufferUnpacker();
    private UnpackerIterator iterator = u.iterator();

    public void onRead(byte[] buffer, int offset, int length) throws IOException {
        // ストリームデシリアライザにデータを供給する
        unpacker.feed(buffer, offset, length);

        // デシリアライズを進める
        while(iterator.hasNext()) {
            Value obj = iterator.next();
            // デシリアライズされたオブジェクトを表示
            System.out.println("deserialized: "+obj);
        }
    }
}
```

利用者は、パイプや TCP ソケットから読み取ったデータを BufferUnpacker::feed メソッドによってストリームデシリアライザに供給していく。そして UnpackerIterator を使用してデシリアライズを行う。

ストリームデシリアライザの実装は、Ruby 実装と同様に、内部バッファを持つ状態遷移機械である。

5.3.2 コピーの削減

OS によって最適化された realloc 関数を利用できない Java では、一度確保したメモリ領域をコピーを発生させずに効率よく拡張することが難しい。メモリ領域を拡張する際にコピーが発生することを防ぐために、MessagePack の Java 実装では、内部バッファの構造を、バッファチャンクのリンクトリストとした。領域の拡張が必要になった場合は、新たなバッファチャンクを確保してリンクトリストにつなぎ合わせる。

さらに Java 実装では、内部バッファにソケットなどから読み取ったデータを供給する際に、コピーを行わずに単にリンクトリストにつなぎ合わせる API を提供した。この利用方法を次に示す。

```
import java.io.IOException
import org.msgpack.MessagePack;
import org.msgpack.type.Value;
import org.msgpack.unpacker.BufferUnpacker;
import org.msgpack.unpacker.UnpackerIterator;

public class IoEventHandler {
    // ストリームデシリアライザをインスタンス化
    private BufferUnpacker unpacker = new MessagePack().createBufferUnpacker();
    private UnpackerIterator iterator = u.iterator();

    public void onRead(byte[] buffer, int offset, int length) throws IOException {
        // フラグをtrueにしてデータを供給すると、データはコピーされずに
        // 内部バッファ内のリンクトリストにつなぎ合わされる
        unpacker.feed(buffer, offset, length, true);

        // デシリアライズを進める
        while(iterator.hasNext()) {
            Value obj = iterator.next();
            System.out.println("deserialized: "+obj);
        }
    }
}
```

このプログラムでは、内部バッファにデータはコピーされず、単にリンクトリストにつなぎ合わせる。これによってデシリアライズ時のコピーを削減している。

5.3.3 直接型変換によるメモリ確保の最適化

Java では、C や C++のようにメモリ空間を直接編集することでオブジェクトを生成するプログラミング手法を利用できないため、メモリ確保に関わる負荷を削減することが難しい。このため、オブジェクト自体を生成しないようにすることが重要になる。

MessagePack の Java 実装では、型変換テンプレートを用いて型変換を行う際に、動的型付けオブジェクトを中間的に生成せずに、直接静的に宣言された型にデシリアライズできるようにする API を提供した。これによって中間オブジェクトの生成コストを削減する。

まず、動的型付けオブジェクトを中間的に生成する利用例を次に示す。

```
import org.msgpack.MessagePack;
import org.msgpack.unpacker.Unpacker;
```

```
import org.msgpack.type.Value;

public class MyReader {
    public static void main(String[] args) throws Exception {
        MessagePack msgpack = new MessagePack();
        Unpacker unpacker = msgpack.createUnpacker(System.in);

        // デシリアライズを行って動的型付けオブジェクトを得る
        Value obj = unpacker.read();  // 中間オブジェクト

        // 静的に宣言された型に型変換を行う
        String converted = msgpack.convert(obj, String.class);
    }
}
```

このプログラムでは、デシリアライズを行って動的型付けオブジェクトを取得し、この動的型付けオブジェクトを静的に宣言された型に変換している。

次に、動的型付けオブジェクトを中間的に生成せずに、直接型変換を行う利用例を次に示す。

```
import org.msgpack.MessagePack;
import org.msgpack.unpacker.Unpacker;
import org.msgpack.type.Value;

public class MyReader {
    public static void main(String[] args) throws Exception {
        MessagePack msgpack = new MessagePack();
        Unpacker unpacker = msgpack.createUnpacker(System.in);

        // デシリアライズと型変換を同時に直接行う
        String converted = unpacker.read(String.class);
    }
}
```

Unpacker.read(Class<T>) メソッドは、内部で動的型付けオブジェクトを中間的に生成することなく、直接指定されたクラスのインスタンスを作成する。これによって中間オブジェクトの生成を防ぎ、GC 負荷を削減した。

5.3.4 動的コード生成によるリフレクションの削減

Java 実装は、4 章で述べた拡張可能な型変換テンプレートを提供する。このとき、型変換テンプレートの拡張を容易にするために、リフレクションを用いてクラス定義から型変換テンプレートを自動生成する機能を実装した。この機能を実現するために、リフレクションを使用した。与えられたクラス定義からメンバフィールドの一覧を読み出すことによって、そのクラスをシリアライズ・デシリアライズするためのプログラムを生成する。

型変換テンプレートの自動生成の使用方法を次に示す。

```
import org.msgpack.MessagePack;
import org.msgpack.packer.BufferPacker;
import org.msgpack.unpacker.BufferUnpacker;
import org.msgpack.type.Value;

public class MyModel {
    // シリアライズされるメンバフィールドを定義する
```

```
    public String name;
    public int age;
    public String address;

    static MessagePack msgpack = new MessagePack();

    static {
        // このクラスをシリアライズするための型変換テンプレートを生成させる
        // ここでリフレクションを使用して、上記のメンバフィールドに関する情報を読み出す
        msgpack.register(MyModel.class);
    }

    public static void main(String[] args) throws Exception {
        MyModel me1 = new MyModel();
        me1.name = "Sadayuki Furuhashi";
        me1.age = 24;
        me1.address = null;

        // シリアライズ
        BufferPacker packer = msgpack.createBufferPacker();
        // それぞれのメンバフィールドをシリアライズする
        packer.write(me1);

        // デシリアライズ
        BufferUnpacker unpacker = msgpack.createBufferUnpacker(packer.toByteArray());
        // それぞれのメンバフィールドをデシリアライズする
        MyModel me2 = unpacker.read(MyModel.class);

        System.out.println("name: "+me2.name);        //~> name: Sadayuki Furuhashi
        System.out.println("age: "+me2.age);          //=> age: 24
        System.out.println("address: "+me2.address);  //=> address: null
    }
}
```

ここで、Java のリフレクションはオーバーヘッドが大きいため、頻繁に実行すると性能が劣化させる要因になる。そこで MessagePack の Java 実装では、まず Java バイトコードの生成を可能にするライブラリである Javassist[16] を用いて、リフレクションで取り出した型に関する情報を用から、型変換を行うバイトコードを生成する。シリアライズやデシリアライズを行う際は、リフレクションを使用せずに単にこのバイトコードを呼び出すことで、リフレクションのオーバーヘッドを削減できるようにした。

5.4 C,C++実装の最適化

5.4.1 メモリゾーンによるメモリ確保の最適化

C や C++では、ポインタを使用してメモリ空間を直接編集することによってオブジェクトを生成することができる。メモリゾーンは、malloc 関数を使用して大きなメモリ領域を一度に確保し、そこから小さなメモリを切り出して利用できるようにする機能である。オブジェクトを解放する際は、この領域を一度に free するだけで、細かく切り出した領域をすべて開放できる。これによってオーバーヘッドの大きい malloc 関数や free 関数の呼び出し回数を削減し、メモリ確保の負荷を削減する。

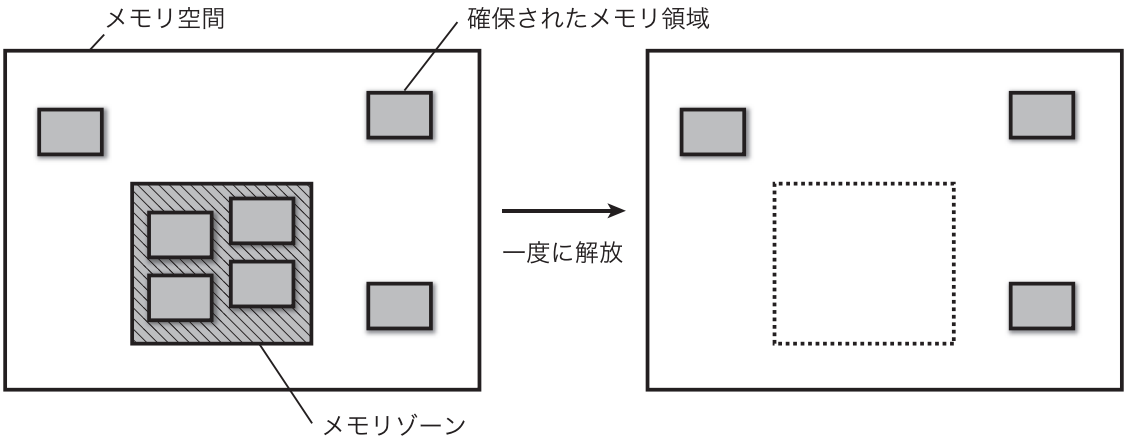

図 5.3: メモリゾーン

実装したメモリゾーンの仕組みを、図 5.3 に示す。C++実装のストリームデシリアライザは、メモリゾーンを使用してオブジェクトの保持に必要なメモリを確保する。

malloc 関数は複数のスレッドから同時に実行されることを想定している一方で、メモリゾーンでは単一のスレッドからのみ利用できるように制限する。通常はデシリアライズを複数のスレッドが同時に行うことはないため、この制限によって使い勝手が損なわれることは無い。これによって排他制御を不要にし、malloc 関数よりも高速にメモリを確保できるようにした。

メモリゾーンを直接利用したプログラムの例を次に示す。

```
/* メモリゾーンを作成 */
/* malloc関数が呼ばれる */
msgpack_zone* z = msgpack_zone_new(MSGPACK_ZONE_CHUNK_SIZE);

/* 5つのメモリ領域を確保して利用 */
char* m1 = msgpack_zone_malloc(z, sizeof(struct s1));
char* m2 = msgpack_zone_malloc(z, sizeof(struct s2));
char* m3 = msgpack_zone_malloc(z, sizeof(struct s3));
char* m4 = msgpack_zone_malloc(z, sizeof(struct s4));
char* m5 = msgpack_zone_malloc(z, sizeof(struct s5));

/* メモリゾーンを解放 */
/* free関数が呼ばれる */
msgpack_zone_free(z);
```

このプログラムと同じ動作を malloc 関数を使って実現したプログラムを次に示す。

```
/* メモリを確保する */
/* サイズはここで固定しなければならない */
char* all = malloc( sizeof(struct s1)+
                    sizeof(struct s2)+
                    sizeof(struct s3)+
                    sizeof(struct s4));

char* m1 = all;
char* m2 = m1 + sizeof(s1);
```

```
char* m3 = m2 + sizeof(s2);
char* m4 = m3 + sizeof(s4);

/* ここでallから追加の領域m5を切り出すことはできない */

/* メモリを解放 */
free(all);
```

このようにメモリゾーンを使用すると、一度に大きなメモリ領域を確保し、そこから小さなメモリを切り出して利用できるようにするプログラミング手法が容易に利用できるようになる。また、利用者は一度に確保するメモリ領域の大きさを意識する必要が無く、プログラムを簡潔にすることができる。メモリゾーンがない場合、利用者はこれらの管理を行うプログラムを記述しなければならない。

第6章　評価

本章では、MessagePack の評価を行う。

6.1　機能評価

6.1.1　異種の言語をまたいだメッセージの交換

MessagePack を用いて、異種のプログラミング言語をまたいでメッセージを交換できることを確認した。

Ruby でメッセージをシリアライズしてファイルに書き出すプログラムを次に示す。

```
require 'msgpack'
data = MessagePack.pack([1,2,3])
File.open('data.msg', 'wb') {|f| f.write data }
```

シリアライズされてファイルに書き出されたメッセージを、Java で読み出してデシリアライズする例を次に示す。

```
import java.io.File;
import java.io.FileInputStream;
import java.io.BufferedInputStream;
import java.io.InputStream;
import org.msgpack.MessagePack;
import org.msgpack.type.Value;

public class Main {
    public static void main(String[] args) throws Exception {
        InputStream in = new BufferedInputStream(
                new FileInputStream(new File("data.msg")));
        Value array = new MessagePack().read(in);
        System.out.println("deserialized: "+array);  //=> deserialized: [1,2,3]
    }
}
```

MessagePack のモデルによって、異種のプログラミング言語をまたいでメッセージを交換できることを確認できた。

6.1.2　移植性

多種のプログラミング言語を横断してメッセージを交換する本研究の目的を達成するためには、他言語への実装の移植が容易でなければならない。現在確認している MessagePack の各言語の実装状況を表 6.1 に示す。

	シリアライザと デシリアライザ	動的型付け 言語	動的型付け オブジェクト	拡張可能な 型変換テンプレート
C	✓		✓	
C++	✓		✓	✓
C#	✓			
D	✓		✓	✓
Go	✓			
Haskell	✓		✓	✓
Java	✓		✓	✓
OCaml	✓		✓	
Scala	✓		✓	✓
Common Lisp	✓	✓		
Erlang	✓	✓		
JavaScript	✓	✓		
Lua	✓	✓		
Objective-C	✓	✓		
JRuby	✓	✓		
Node.JS	✓	✓		
Perl	✓	✓		
PHP	✓	✓		
Python	✓	✓		
Ruby	✓	✓		
Smalltalk	✓	✓		

表 6.1: MessagePack の各言語の実装状況

それぞれの実装は、必ずしも MessagePack のすべての機能を実装していない。しかし、シリアライザとデシリアライザはすべての言語で実装されている。4 章で述べたように、MessagePack のすべての機能を実装することは難しいが、MessagePack を最低限利用可能にする実装は簡単であることが、多言語対応を容易にしていることを確認した。

また、多くのプログラミング言語をまたいで利用することを目的としたシリアライズ形式を含んだシステムである Avro および Protocol Buffers について、それらを利用可能な言語の一覧を以下に示す。

Avro C, Java, Python

Protocol Buffers C++, Java, PHP, Python, Ruby

MessagePack は、これらのシリアライズ形式と比較して対応している言語が多い。MessagePack によってより多くのプログラミング言語を横断してメッセージを交換することが可能になり、本研究の目的が達成された。

6.1.3 動的型付けオブジェクト API

MessagePack では、MessagePack の型システムに基づいた動的型付けオブジェクト API を提供することによって、4.1 節で述べたように、静的型付け言語においてもオブジェクトの型を利用した柔軟なプログラミングが可能にした。このような動的型付けオブジェクト API は、他のシリアライズシステムには存在しない。

6.1.4 型変換テンプレート

MessagePack では、デシリアライズされた動的型付けオブジェクトを静的に宣言された型に変換する API をライブラリとして提供した。これにより、4.2 節で述べたように、型検査を行うプログラムを自分で記述することなく、デシリアライズされたオブジェクトを静的な型に変換して安全に扱うことが可能になった。このように動的型付けオブジェクトを静的な型に変換する体系化された機構は、他のシリアライズシステムには存在しない。

6.2 性能評価

6.2.1 C,C++実装の性能評価

メモリゾーンの効果

C 実装のメモリゾーンの実装の性能を確認するため、C 実装のメモリゾーンの性能を測定した。メモリゾーンを使ってメモリを確保する速度と、malloc 関数を使ってメモリを確保する速度をそれぞれ測定した。128 バイトのメモリを 100 万回確保する速度を計測することで、

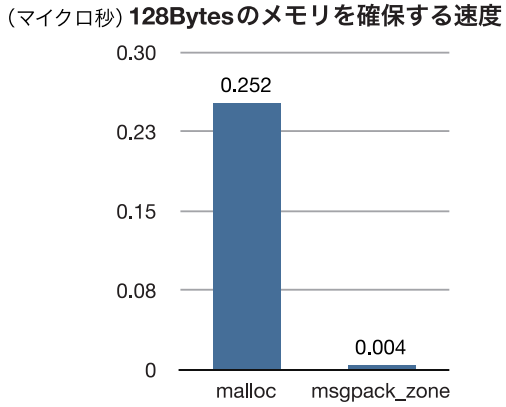

図 6.1: malloc 関数とメモリゾーンの比較

128 バイトのメモリを 1 回確保するのにかかる時間を算出した。測定に用いたハードウェアおよびソフトウェアを次に示す

CPU AMD Athlon64 X2 5000+

メモリ DDR2 8GB

malloc の実装 glibc-2.8

得られた結果を図 6.1 に示す。メモリゾーンは、malloc 関数と比べて 60 倍以上高速であることが分かる。これは、malloc 関数は複数のスレッドから実行されることを含めて、汎用的に実装されていなければならないのに対し、メモリゾーンは単一のスレッドからのみ利用でき、MessagePack の内部でのみ利用できれば十分であることから、排他制御などの処理を省略しているためだと考えられる。

6.2.2 Ruby 実装の性能評価

Ruby 実装の最適化の効果を確認するため、Ruby の処理系に同梱されている JSON 処理系と性能を比較した。測定に用いたハードウェアを次に示す。

CPU Intel Core i7 1.8GHz

メモリ DDR3 4GB

OS Mac OS X 10.7.2

Ruby 処理系 ruby 1.9.2p290

”1”から”4194304”までの連番の文字列を含む配列オブジェクトを JSON と MessagePack を使用してシリアライズ・デシリアライズする処理にかかった時間と、シリアライズされたデータのサイズを計測した。得られた結果を図 6.2 に示す。

JSON と比較すると、シリアライズとデシリアライズにかかる時間は 5 分の 1 に以下であった。実装の最適化が効果を発揮していると考えられる。またシリアライズ後のデータサイズ

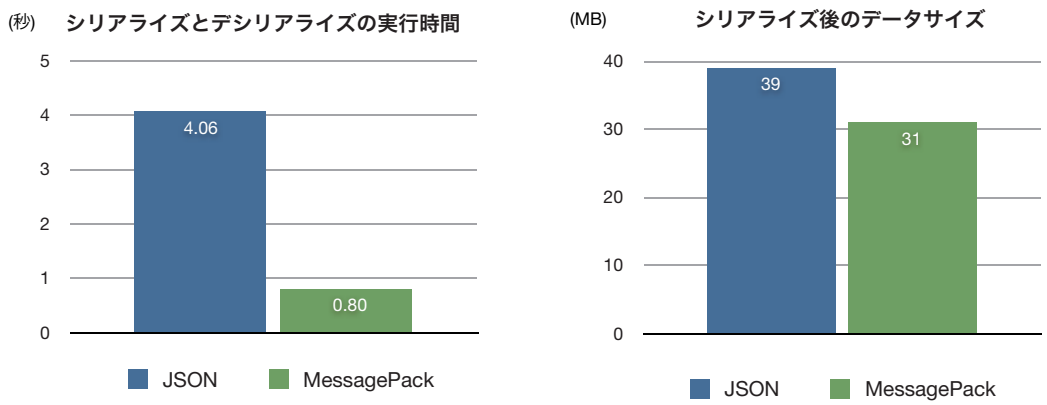

図 6.2: Ruby 版の性能評価

は、JSON と比較して約 20%小さい。MessagePack は JSON と同様にシリアライズされたオブジェクトの型に関する情報を格納するが、よりコンパクトに格納できるようにする表現形式の設計が効果を発揮していると考えられる。

6.2.3 Java 実装の性能評価

Java 実装の最適化の効果を確認するため、シリアライズおよびデシリアライズの性能を測定した。測定には、JVM 上で動作するシリアライズライブラリの性能測定ツールである jvm-serializers[17] を使用し、Avro、Protocol Buffers、Thrift および JSON との性能を比較した。jvm-serializers は、整数、文字列、浮動小数点数などからなるオブジェクトを繰り返しシリアライズ・デシリアライズするためにかかった時間を測定することで、あるシリアライズライブラリが 1 つのオブジェクトを 1 回シリアライズ・デシリアライズするのに必要な時間を算出する。測定に用いたハードウェアおよびソフトウェアを次に示す。

CPU Intel Core i7 1.8GHz
メモリ DDR3 4GB
OS Mac OS X 10.7.2

得られた結果を図 6.3 と図 6.4 と示す。

処理時間では、他のシリアライズライブラリと比べても高速であることが確認できた。また、シリアライズ後のデータサイズは、型情報をシリアライズ後のデータの中に含めないシリアライズ形式（Avro, Protocol Buffers, Thrift）と比較しても、十分小さいことが確認できた。Avro はシリアライズ後のデータの先頭に IDL を付与することができるが、この性能評価では付与されていない。

実装の性能最適化によって、効率の良いメッセージ交換手法を提供する本研究の目的を達成できた。

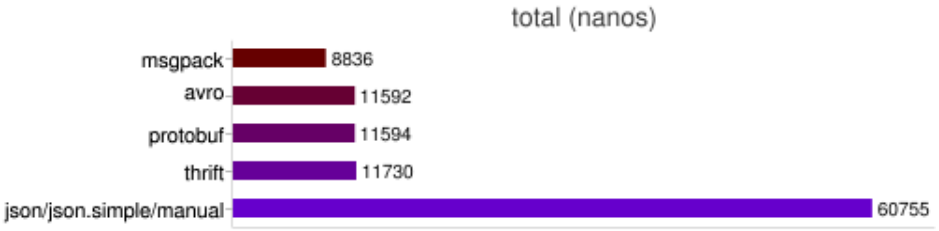

図 6.3: シリアライズとデシリアライズの実行時間 (単位:ナノ秒)

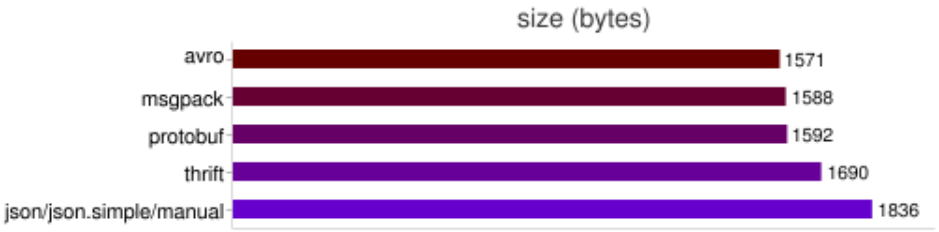

図 6.4: シリアライズ後のデータの大きさ (単位:バイト)

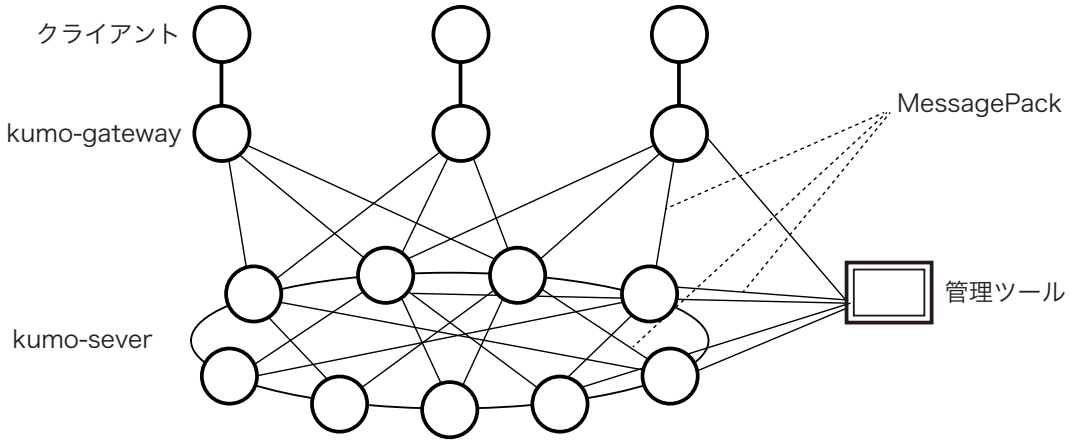

図 6.5: kumofs

6.3 アプリケーションの開発

MessagePack を使用して実際的なアプリケーションを設計、開発、運用および継続的な更新をすることによって、異種のプログラミング言語・異種の計算機環境・異種のバージョン間で互換性を維持しながら利用可能であり、高い相互運用性を実現できたことを確認した。

6.3.1 kumofs

kumofs[18] は、高い耐障害性と並列性を重視した分散データストアである。Consistent Hashing による分散とレプリケーションを実装している。C++と Ruby の 2 つの言語を用いて実装し、ノード間の通信プロトコルに MessagePack を使用した。アーキテクチャを図 6.5 に示す。

MessagePack の動的型付けの特性を利用して、過去のバージョンと通信プロトコルの互換性を維持したまま、ソフトウェアのバージョンアップを可能とした。これに加え、kumofs はシステムに参加するノードの一部が故障しても正常な動作を継続できる耐障害性を備えていることから、ノードを一度に再起動するのではなく次々に再起動していくことで、システム全体を停止させることなくソフトウェアのバージョンアップを行うことが可能になった。

また、kumofs はデータストアであることから、比較的サイズの大きいデータを扱うが、MessagePack の C++実装の最適化により、プログラム内でコピーすることなく効率的に扱うことができる。これによって高い性能を発揮することが可能になった。

さらに kumofs は、複数の言語の特性を活かして実装した。データの保存と検索を行うサーバ群は C++で実装したが、それらを一括して管理するためのツール群は Ruby で実装した。これにより、サーバ群は極めて高速に動作させることができた一方で、管理ツール群の実装と拡張が容易になった。

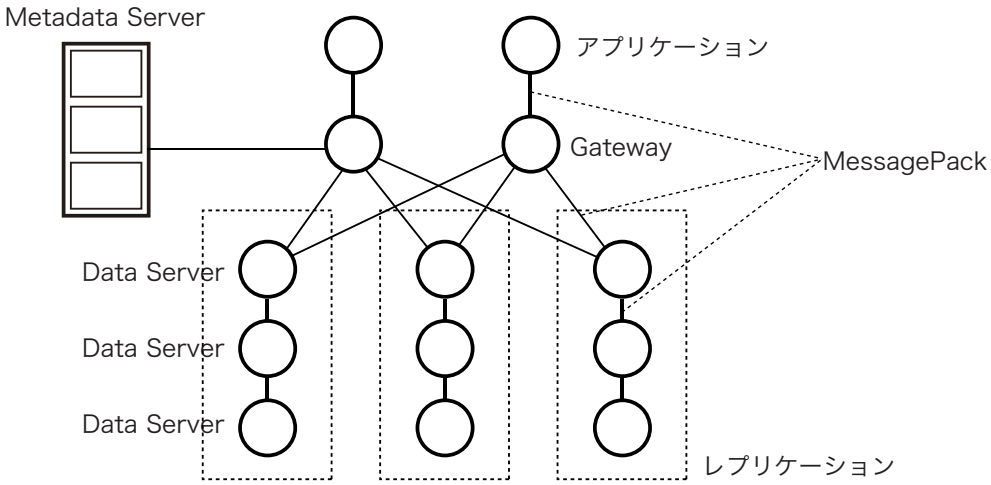

図 6.6: LS4

kumofs では、MessagePack を利用することによって無停止でシステムのバージョンアップを可能にし、また高い性能を発揮することができた。また、多言語間通信が可能であることから、2 つの言語の利点を組み合わせることで実装の手間を大幅に簡略化することができた。

6.3.2 LS4

LS4[19] は、高い耐障害性とスループットを重視した分散ストレージシステムである。ロードバランスとマスタ・スレーブ型のレプリケーションを実装している。Ruby を用いて実装し、ノード間の通信プロトコルとメタデータの表現形式に MessagePack を用いた。アーキテクチャを図 6.6 に示す。

LS4 は、Web アプリケーションから直接利用されることを想定したシステムであることから、スクリプト言語を含む多くの言語から利用できる通信プロトコルを採用する必要があった。一方で、写真や動画などのサイズの大きいデータを扱うシステムであることから、バイナリ形式で効率の良い通信プロトコルが必要であった。MessagePack を使用することで、これらの条件を容易に満たすことができた。

また LS4 では、高い耐障害性を実現するためにレプリケーション機能を実装したが、レプリケーション・ログの表現形式に MessagePack を用いた。プログラムの更新に伴って、レプリケーション・ログの構造を変更しなければならない機会があったが、MessagePack の動的型付けの柔軟性を使用することによって、過去に作成されたレプリケーション・ログとの互換性を維持したまま更新することが可能であった。

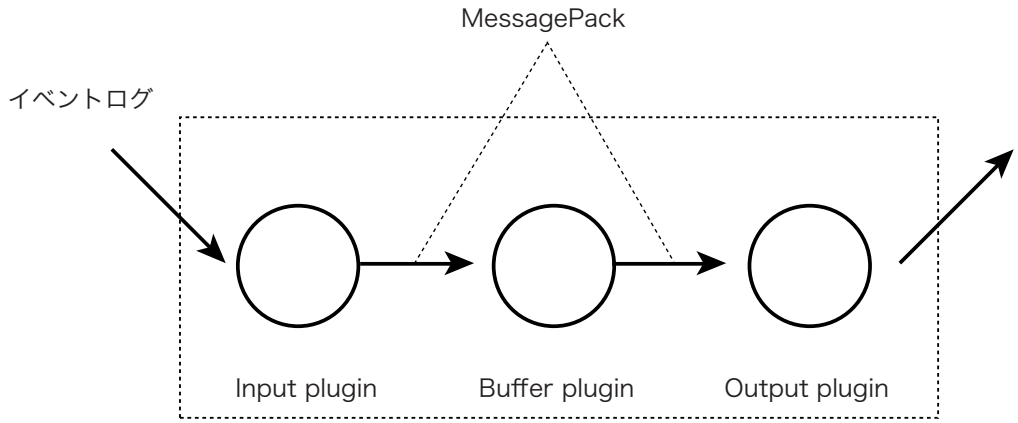

図 6.7: Fluentd

6.3.3 Fluentd

Fluentd[20] は、プラグイン構造を採用したイベントログ収集ツールである。Ruby を用いて実装し、プラグイン間のインタフェースに MessagePack を用いた。アーキテクチャを図 6.7 に示す。

Fluentd は、入力、出力、そしてバッファリングの 3 種類のプラグインを組み合わせることによって、様々な目的で利用できるという特徴を持つツールである。Fluentd の中核は、それらのプラグイン間の連携を調停する枠組みである。このようなプラグイン構造を採用するプログラムでは、プラグイン間のインタフェースの設計が拡張可能性を大きく左右するが、Fluentd は MessagePack を利用することによって、高いスループットと API のシンプルさを実現することができた。また、様々なプログラミング言語を用いてプラグインを開発することが可能になった。

さらに、MessagePack の型システムは JSON の型システムと互換性が高いことから、他のシステムとのインタフェースとして MessagePack の代わりに JSON を選択可能にすることにより、型システムを相互変換するプログラムを実装することなく相互運用性を高めることが可能になった。

第7章　おわりに

本論文では、分散システムのためのメッセージ表現手法である MessagePack について述べた。

第 1 章では、本研究の背景および概要について述べた。今日では、多種多様なサービスをネットワークを通じて多数のユーザーに提供するシステムの構築方法が一般的となっている。これらのシステムを実装するには、異種混合の計算環境においても高い相互互換性を実現する、効率の良いメッセージ表現手法が必要とされていることを提示した。また、そのソフトウェアライブラリとしての使い勝手も重要になる。そこで本研究の目的は、異種のプログラミング言語・異種の計算機環境・異種のバージョン間で互換性を維持しながら利用可能であり、高速に動作するメッセージ表現手法を設計および実装することで、分散システムの開発を容易にすることに設定した。

第 2 章では、本研究に関連した先行研究について述べた。XDR、SunRPC、JSON、Thrift、Protocol Buffers、Avro、JavaRMI、および CORBA ついて、その概要と課題について述べた。

第 3 章では、MessagePack の設計について述べた。

まず、メッセージ交換手法の設計における課題について述べた。MessagePack では、どのプログラミング言語にも依存しない、中立的な型システムを定義し、この型システムとプログラミング言語の型システムを相互に変換する仕組みを実装することで、多数のプログラミング言語を横断してメッセージを交換することを可能にした。また、シリアライズ後のデータの中にオブジェクトの型を表す情報を含めることで、シリアライズ後のデータの互換性を損なわずに、そのデータを扱うプログラムを更新することを可能にした。このとき、シリアライズ後のデータサイズが増大してしまう問題があるが、MessagePack ではより頻繁に利用されると想定される型に対して多くのビットを割り当てることにより、型に関する情報をコンパクトに格納する。

第 4 章では、MessagePack の言語バインディングについて述べた。その特徴は、静的な型付けを採用するプログラミング言語においても、MessagePack の型システムに基づいた動的な型付けを利用可能にした点にある。また同時に、型変換テンプレートと呼ぶ機構を導入することによって、MessagePack の型システムと各言語の型システムの相互変換を可能にした。

第 5 章では、MessagePack の実装について述べた。各言語の実装について、コピーを削減して性能を向上させるための最適化を行った。また、オブジェクトの生成コストを削減することによって性能を向上させ、また GC 負荷を削減する最適化を行った。

第 6 章では、MessagePack の評価を行った。

MessagePack は、多言語対応を目的の一つした他のシリアライズ形式と比較して対応している言語が非常に多く、その移植性が高いことを示した。MessagePack のすべての機能を実

装することは難しいが、MessagePack を最低限利用可能にする実装は簡単であることが、多言語対応を容易にした。

また、C++、Java、Ruby の各実装について、その性能を測定した。他のシリアライズ形式と比較しても高速に動作することを確認し、実装の最適化によって、高速に動作するメッセージ交換手法を提供する本研究の目的を達成できたことを確認した。

最後に、MessagePack を使用して実際に開発したアプリケーションについて述べた。MessagePack によってメモリや CPU などの資源の消費量が少なく、高速に動作し、多種のプログラミング言語を横断して利用することができ、異種混合の計算機環境においても高い相互互換性を実現するメッセージ表現手法を容易に利用できるようになったことから、分散システムの実装が容易になったことを確認した。

現在、MessagePack はオープンソースコミュニティによって改善が進められている。各国の開発者によって 20 以上のプログラミング言語で実装され、様々なサービスで利用されている。今後もシステムの設計者として開発を先導していく。

謝辞

本研究を行うにあたり、指導教員になっていただき、終始丁寧なご指導、貴重な助言をいただきました新城靖准教授に心より感謝申し上げます。加藤和彦教授、前田敦司教授、板野肯三教授、佐藤聡講師、ならびに中井央准教授には多くのご指導と助言を承りました。深く感謝申し上げます。またこの論文をまとめるにあたり、木村成伴准教授に多大な協力を頂きました。心から感謝申し上げます。また、MessagePack の開発に参加していただいた各国のコミッタの皆様、そして日々の研究においてお世話になりましたソフトウェア研究室の皆様に、感謝申し上げます。最後に、MessagePack をビジネスに活用しながらオープンソースソフトウェアとしてその開発に注力することを可能にしてくれた Treasure Data, Inc. の共同設立者、同僚そして投資家の皆様に、この場を借りて御礼を述べさせていただきます。

参考文献

[1] Mark Slee, Aditya Agarwal and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. http://incubator.apache.org/thrift/static/thrift-20070401.pdf.

[2] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–8, 2007.

[3] 古橋貞之, 新城靖, 佐藤聡, 中井央, 板野肯三. 効率的な多言語間オブジェクト交換手法の設計と実装. 情報処理学会　第 21 回コンピュータシステム・シンポジウムポスターデモセッション, 2009.

[4] M. Eisler. XDR: External Data Representation Standard. RFC 4506, May 2006.

[5] R. Thurlow. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 5531, May 2009.

[6] Apache Thrift, 2010 年 2 月 19 日閲覧. http://incubator.apache.org/thrift/.

[7] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*, Vol. 13, pp. 277–298, 2005.

[8] Apache Avro, 2010 年 2 月 19 日閲覧. http://hadoop.apache.org/avro/.

[9] Troy Bryan Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1998.

[10] Michi Henning. The rise and fall of corba. *Queue*, Vol. 4, No. 5, pp. 28–34, 2006.

[11] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, Vol. 40, No. 9, pp. 1098–1101, September 1952.

[12] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, July 2006.

[13] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. August 2008.

[14] Wireshark: A Network Protocol Analyzer, 2012 年 1 月 13 日閲覧. http://www.wireshark.org/.

[15] Wireshark Plugin for MessagePack-RPC, 2012 年 1 月 13 日閲覧. https://github.com/kumagi/dissector-msgpack-rpc.

[16] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pp. 364–376, 2003.

[17] jvm-serializers, 2012 年 1 月 13 日閲覧.

[18] Sadayuki Furuhashi. kumofs, 2010 年 2 月 19 日閲覧. http://github.com/etolabo/kumofs.

[19] Sadayuki Furuhashi. The LS4 Project — open source distributed storage system, 2012 年 1 月 13 日閲覧. http://ls4.sourceforge.net/.

[20] Sadayuki Furuhashi. Fluentd, 2012 年 1 月 13 日閲覧. http://fluentd.org/.

付 録A　MessagePackの表現形式

MessagePack で扱うデータ型の表現形式について示す。

複数のバイトからなるデータを保存する場合は、エンディアンが問題になる。MessagePack では、ネットワークバイトオーダーを用いる。

ここでは次のような表記を用いて表現形式を表す。以下に示す箱は、1 バイトを表す。

```
+--------+
|        |
+--------+
```

次に以下す箱は、可変長バイトを表す。

```
+================+
|                |
+================+
```

A.1　Nil

Nil は、0xc0 で表す。

```
+--------+
|  0xc0  |
+--------+
```

A.2　Boolean

Boolean は、True か False を表す。True は、0xc3 で表す。

```
+--------+
|  0xc3  |
+--------+
```

False は、0xc2 で表す。

```
+--------+
|  0xc2  |
+--------+
```

A.3 Integer

Integer は、10 種類の表現形式を持つようにした。

A.3.1 Positive Fixnum

Positive Fixnum は、[0, 127] の範囲の整数を 1 バイトで保存する。1 バイトのうち、先頭の 1 ビットは 0 で、それ以降のビットで整数値を表す。

```
+--------+
|0XXXXXXX|
+--------+
```

A.3.2 Negative Fixnum

Negative Fixnum は、[-32, -1] の範囲の整数を 1 バイトで保存する。1 バイトのうち、先頭の 3 ビットは 111 で、それ以降のビットで整数値を表す。

```
+--------+
|111XXXXX|
+--------+
```

A.3.3 uint 8、uint 16、uint 32、uint 64

uint 8、uint 16、uint 32、uint64 は、それぞれ符号無し 8 ビット、16 ビット、32 ビット、64 ビットの整数を、それぞれ 2 バイト、3 バイト、5 バイト、9 バイトで保存する。先頭の 1 バイトは、それぞれ 0xcc、0xcd、0xce、0xcf で、それ以降のバイトで整数値を表す。

```
unsigned 8-bit interger
+--------+--------+
|  0xcc  |XXXXXXXX|
+--------+--------+

unsigned 16-bit integer
+--------+--------+--------+
|  0xcd  |XXXXXXXX|XXXXXXXX|
+--------+--------+--------+

unsigned 32-bit integer
+--------+--------+--------+--------+--------+
|  0xce  |XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|
+--------+--------+--------+--------+--------+

unsigned 64-bit integer
+--------+--------+--------+--------+--------+--------+--------+--------+--------+
|  0xcf  |XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|
+--------+--------+--------+--------+--------+--------+--------+--------+--------+
```

A.3.4 int 8、int 16、int 32、int 64

int 8、int 16、int 32、int64 は、それぞれ符号付き 8 ビット、16 ビット、32 ビット、64 ビットの整数を、それぞれ 2 バイト、3 バイト、5 バイト、9 バイトで保存する。先頭の 1 バイトは、それぞれ 0xd0、0xd1、0xd2、0xd3 で、それ以降のバイトで整数値を表す。

```
signed 8-bit integer
+--------+--------+
|  0xd0  |XXXXXXXX|
+--------+--------+

signed 16-bit integer
+--------+--------+--------+
|  0xd1  |XXXXXXXX|XXXXXXXX|
+--------+--------+--------+

signed 32-bit integer
+--------+--------+--------+--------+--------+
|  0xd2  |XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|
+--------+--------+--------+--------+--------+

signed 64-bit integer
+--------+--------+--------+--------+--------+--------+--------+--------+--------+
|  0xd3  |XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|
+--------+--------+--------+--------+--------+--------+--------+--------+--------+
```

A.4 Float

Float は、2 種類の表現形式を持つようにした。

A.4.1 float

float は、IEEE 754 形式の単精度二進化浮動小数点数 [13] を 5 バイトで保存する。先頭の 1 バイトは 0xca で、それ以降のバイトで浮動小数点数を表す。

```
+--------+--------+--------+--------+--------+
|  0xca  |XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|
+--------+--------+--------+--------+--------+
```

A.4.2 double

double は、IEEE 754 形式の倍精度二進化浮動小数点数 [13] を 9 バイトで保存する。先頭の 1 バイトは 0xcb で、それ以降のバイトで浮動小数点数を表す。

```
+--------+--------+--------+--------+--------+--------+--------+--------+--------+
|  0xcb  |XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|
+--------+--------+--------+--------+--------+--------+--------+--------+--------+
```

A.5 Raw

Raw は、3 種類の表現形式を持つようにした。

A.5.1 FixRaw

FixRaw は、最大 31 バイトのバイト列を、(バイト列の長さ+1) バイトで保存する。先頭の 1 バイトでバイト列の長さを表し、それ以降にバイト列を保存する。先頭の 1 バイトのうち、先頭の 3 ビットは 101 で、それ以降のビットでバイト列の長さを表す。

```
+--------+=================+
|101XXXXX|    N bytes      |
+--------+=================+
```

A.5.2 raw 16、raw 32

raw 16、raw 32 は、それぞれ最大 $(2^{16}) - 1$ バイト、最大 $(2^{32}) - 1$ バイトのバイト列を、それぞれ (バイト列の長さ+3) バイト、(バイト列の長さ+5) バイトで保存する。先頭の 1 バイトはそれぞれ 0xda、0xdb で、続く 2 バイトまたは 4 バイトでバイト列の長さを表す。

```
+--------+--------+--------+=================+
|  0xda  |XXXXXXXX|XXXXXXXX|    N bytes      |
+--------+--------+--------+=================+

+--------+--------+--------+--------+--------+=================+
|  0xdb  |XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|    N bytes      |
+--------+--------+--------+--------+--------+=================+
```

A.6 Array

Array は、3 種類の表現形式を持つようにした。

A.7 FixArray

FixArray は、最大 15 要素の配列を保存する。先頭の 1 バイトで要素数を表し、後続する要素数個分のオブジェクトを配列の要素とする。先頭の 1 バイトのうち、先頭の 4 ビットは 1001 で、後続するビットで要素数を表す。

```
+--------+=================+
|1001XXXX|   N objects     |
+--------+=================+
```

A.7.1 array 16、array 32

array 16、array 32 は、それぞれ最大 $(2^{16}) - 1$ 要素、$(2^{32}) - 1$ 要素の配列を保存する。先頭の 1 バイトはそれぞれ 0xdc、0xdd で、続く 2 バイトまたは 4 バイトで要素数を表し、後続する要素数個分のオブジェクトを配列の要素とする。

```
+--------+--------+--------+================+
|  0xdc  |XXXXXXXX|XXXXXXXX|   N objects    |
+--------+--------+--------+================+

+--------+--------+--------+--------+--------+================+
|  0xdd  |XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|   N objects    |
+--------+--------+--------+--------+--------+================+
```

A.8 Map

Map は、3 種類の表現形式を持つようにした。

A.8.1 FixMap

FixMap は、最大 15 要素の連想配列を保存する。先頭の 1 バイトで要素数を表し、後続する (要素数× 2) 個のオブジェクトを連想配列の要素とする。奇数番目のオブジェクトは連想配列のキーを表し、その次のオブジェクトはそのキーに対応する値を表す。先頭の 1 バイトのうち、先頭の 4 ビットは 1000 で、後続するビットで要素数を表す。

```
+--------+=================+
|1000XXXX|  N*2 objects    |
+--------+=================+
```

A.8.2 map 16、map 32

map 16、map 32 は、それぞれ最大 $(2^{16}) - 1$ 要素、$(2^{32}) - 1$ 要素の連想配列を保存する。先頭の 1 バイトはそれぞれ 0xde、0xdf で、続く 2 バイトまたは 4 バイトで要素数を表し、後続する (要素数×)2 個のオブジェクトを連想配列の要素とする。奇数番目のオブジェクトは連想配列のキーを表し、その次のオブジェクトはそのキーに対応する値を表す。

```
+--------+--------+--------+================+
|  0xde  |XXXXXXXX|XXXXXXXX|   N*2 objects  |
+--------+--------+--------+================+

+--------+--------+--------+--------+--------+================+
|  0xdf  |XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|   N*2 objects  |
+--------+--------+--------+--------+--------+================+
```