平成30年度　修 士 論 文

和文題目

アプリケーションから機能拡張された
通信ライブラリを利用する方法に関する研究

英文題目

A Study on
How to Use an Extended Communication Library
from Applications

情報工学専攻　渡邊研究室

(学籍番号: 173426012)

清水　一輝

提出日: 平成31年01月29日

名城大学大学院理工学研究科

概要

ライブラリは処理速度や移植性，他言語との連携といった観点から，C 言語で実装される．しかし，スマートフォン用アプリケーションはC 言語よりも抽象度の高い Java や Swift といった高級言語によって開発されることが一般的である．そのため，C 言語ライブラリを使用したい場合は，一般的に高級言語に応じたラッパーを経由し，C 言語ライブラリを意識したプログラミングが必要である．しかし，通信ライブラリのようなプログラミング言語の標準 API (Application Programming Interface) として備わっている機能を拡張するライブラリの場合は，高級言語の提供する標準 API と同じ使用方法を踏襲できることが望ましい．

本研究では，スマートフォン向けアプリケーションに通信ライブラリを適用する方法をラッパー方式と VPN (Virtual Private Network) 方式といった 2 通りの提案と一部実装を行った．ラッパー方式はアプリケーションに組み込むことで，機能拡張された通信を行うための方式である．また，VPN 方式は Android OS/iOS の VPN 機能を利用することで，アプリケーションに機能拡張された通信を提供する方式である．動作検証を行った結果，提案方式を用いてアプリケーションから C 言語通信ライブラリを利用可能であることを確認した．

Abstract

The library is implemented in programming language C from the viewpoint of processing speed, portability, and cooperation with other languages. However, applications for smartphones are generally developed by high-level languages such as Java and Swift, which are more abstract than C. Therefore, when you want to use the C library, programming that considers the C library generally is necessary via the wrapper corresponding to the high-level language. However, in the case of a library that extends functions provided as a standard API (Application Programming Interface) of a programming language such as a communication library, it is desirable to be able to follow the same usage method as the standard API provided by a high-level language.

In this research, we have implemented two ways of applying communication libraries to smartphone applications, and some implementations such as wrapper method and VPN (Virtual Private Network) method. The wrapper method is a method for performing extended communication by incorporating it in an application. In addition, the VPN method provides a function extended function to applications by using the VPN function of Android OS / iOS. As a result of the operation verification, we confirmed that the C communication library can be used from the applications using the proposed method.

目 次

第1章　序論

アプリケーションやプログラムで使用されるライブラリは処理速度や移植性，他言語との連携といった観点から，C 言語によって実装されることが多い．一方，スマートフォン用アプリケーションや Web アプリケーションでは，C 言語よりも抽象度の高い Java や Swift，JavaScript 等といった高級言語で開発されることが一般的である．そのため，アプリケーション開発時に C 言語ライブラリを使用したい場合は，一般的にアプリケーションの開発言語に応じたラッパーを作成する必要がある．そして，アプリケーションはラッパーを使用し，経由することによって C 言語ライブラリを利用できるようになる．

既存のラッパー生成技術である SWIG (Simplified Wrapper and Interface Generator) [1–3] では，C 言語ライブラリ用に SWIG 用の独自記述ファイルを作成することで，高級言語用のラッパーを生成することができる．しかし，SWIG によって生成されるラッパーは，API (Application Programming Interface) が C 言語ライブラリの実装に依存するという課題がある．この課題は，画像の特徴点抽出等といった高級言語には存在しない機能を C 言語ライブラリ側で提供する場合であれば問題はない．しかし，暗号化通信のような高級言語に存在する既存の通信機能を拡張した機能であれば，これらの機能は高級言語と同じ使用方法であることが望ましい．

また，通信方式をスマートフォンユーザに提供することで，ユーザが任意のスマートフォンアプリケーションに対して通信ライブラリを適用したい場合がある．この場合，ラッパーは予めアプリケーションに組み込む必要があるため，ユーザはラッパーを使用してアプリケーションに通信ライブラリを適用することは不可能である．そのため，アプリケーションが生成したパケットを書き換えることで，アプリケーションに一切変更を加えることなく通信ライブラリを適用する方法を提供することが望ましい．

本研究では，アプリケーションがラッパーを使用することで C 言語通信ライブラリの通信を適用するラッパー方式と Android OS/iOS の提供する VPN (Virtual Private Network) 機能を利用することでスマートフォン内のアプリケーションに C 言語通信ライブラリの通信を適用させる VPN 方式を提案する．提案方式を適用する C 言語通信ライブラリには，NTMfw (NTMobile framework library) [4–6] を使用した．NTMfw は，移動透過性と通信接続性を同時に実現する技術である NTMobile (Network Traversal with Mobility) [7–9] を C 言語ライブラリ化したものである．ラッパー方式では，NTMfw が提供するソケット API を高級言語のソケット API と同じ使用方法で使用可能にする．VPN 方式では，アプリケーションが生成したパケットを NTMfw の機能を使用して書き換えることで，NTMfw による通信を実現する．Java を対象としたラッパー方式の実装と iOS を対象とした VPN 方式の一部実装を行い，動作検証及び性能評価を行った．動作検証を行った結果から，実装したラッパーと VPN アプリケーションが実現可能であることを確認した．また，性能評価を行った結果から，

ラッパー方式よりも VPN 方式の方が処理時間が優れていると考えられる．

以後，2 章では既存のラッパー生成技術である SWIG と C 言語や Java，Node.js といった各プログラミング言語の UDP ネットワークプログラミングについて，3 章では提案するラッパー方式と VPN 方式について述べる．4 章では提案方式を適用する通信ライブラリである NTMfw について述べ，5 章では提案方式の実装について述べる．6 章ではラッパー方式と VPN 方式の土台となる TUN 利用型の動作検証や性能評価，比較を行い，最後に 7 章でまとめる．

第2章　既存技術

本章では，既存のラッパー生成技術である SWIG について述べる．また，C 言語，Java，Node.js といった 3 種類のプログラミング言語に標準で搭載されている UDP (User Datagram Protocol) 通信用 API を比較し，各言語の特徴について述べる．

2.1 SWIG

SWIG[*1]とは，C 言語や C++で書かれたプログラムやソフトウェアを高級言語と連携させることで，高級言語から C/C++プログラムやソフトウェアを使用可能にするソフトウェア開発用ツールである．SWIG により扱うことのできる高級言語は，JavaScript や Perl，PHP，Python，Ruby，Tcl/Tk，Lua，Octave，Scilab，R，C#，CommonLisp (CLISP，Allegro CL，CFFI，UFFI)，D，Go 言語，Modula-3，OCAML などがある．

図 1 に SWIG の概要を示す．SWIG は，C/C++ライブラリとそれに対応した SWIG 用の独自記述ファイルである i ファイルを用いて，C 言語もしくは C++と高級言語を連携させるラッパー用グルーコードを生成する．その後，ラッパー用グルーコードから連携したい高級言語用ラッパーファイルを生成する．この高級言語用ラッパーファイルを高級言語アプリケーションが使用することで，高級言語から C 言語もしくは C++のプログラムやライブラリを使用できるようになる．ラッパー用グルーコードと高級言語用ラッパーファイルの生成は SWIG により自動で行われるが，独自記述ファイルである i ファイルは開発者が新たに作成する必要がある．

SWIG によって自動生成されたラッパーは，C 言語もしくは C++のプログラムやライブラリの実装に依存するという課題がある．この課題は，画像の特徴点抽出などといった一般的に高級言語には存在しない新機能の場合であれば問題はない．しかし，暗号化/復号を行う通信のように一般的に高級言語に存在する既存の通信機能を拡張した機能であれば，これらの機能の使用方法は既存の通信機能と同じ使用方法であることが望ましい．

以下に，C 言語で実装された”Hello World!” を出力する `hello()` 関数を Java から使用するサンプルコードを示す．ソースコード 2.1 に `hello()` 関数を実装した C プログラム，ソースコード 2.2 に C プログラム用に作成した i ファイルを示す．また，ソースコード 2.3 に C プログラムと i ファイルを用いて SWIG で生成したラッパー用グルーコードの一部，ソースコード 2.4 にラッパー用グルーコードから生成した Java 用ラッパーファイルを示す．ソースコード 2.5 に Java 用ラッパーを使用して `hello()` 関数を実行する Java プログラムを示す．

[*1] https://www.swig.org/

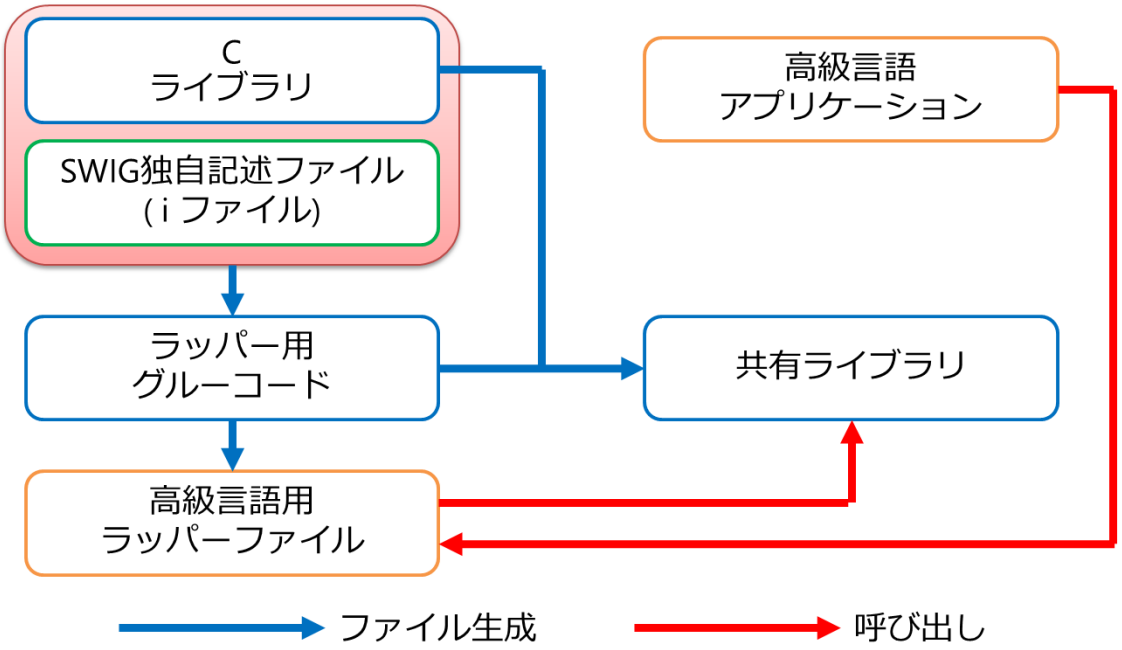

図 1　SWIG の概要

ソースコード 2.1　C プログラム

```
1 #include <stdio.h>
2
3 void hello (){          // Hello World!を出力するhello()関数
4         printf("Hello␣World!\n");
5 }
```

ソースコード 2.2　SWIG 用独自記述ファイル (i ファイル)

```
1 %module hello
2 %{
3 #include "stdio.h"
4 %}
5 void hello();
```

ソースコード 2.3　ラッパー用グルーコード

```
1  #define SWIGJAVA
2
3  #include <jni.h>
4  #include <stdlib.h>
5  #include <string.h>
6
7  /* Support for throwing Java exceptions */
8  typedef enum {
9          SWIG_JavaOutOfMemoryError = 1,
10         SWIG_JavaIOException,
11         SWIG_JavaRuntimeException,
```

```
12          SWIG_JavaIndexOutOfBoundsException,
13          SWIG_JavaArithmeticException,
14          SWIG_JavaIllegalArgumentException,
15          SWIG_JavaNullPointerException,
16          SWIG_JavaDirectorPureVirtual,
17          SWIG_JavaUnknownError
18  } SWIG_JavaExceptionCodes;
19
20  typedef struct {
21          SWIG_JavaExceptionCodes code;
22          const char *java_exception;
23  } SWIG_JavaExceptions_t;
24
25  static void SWIGUNUSED SWIG_JavaThrowException(JNIEnv *jenv, SWIG_JavaExceptionCodes code
        , const char *msg) {
26          jclass excep;
27          static const SWIG_JavaExceptions_t java_exceptions[] = {
28                  { SWIG_JavaOutOfMemoryError, "java/lang/OutOfMemoryError" },
29                  { SWIG_JavaIOException, "java/io/IOException" },
30                  { SWIG_JavaRuntimeException, "java/lang/RuntimeException" },
31                  { SWIG_JavaIndexOutOfBoundsException, "java/lang/
                        IndexOutOfBoundsException" },
32                  { SWIG_JavaArithmeticException, "java/lang/ArithmeticException" },
33                  { SWIG_JavaIllegalArgumentException, "java/lang/IllegalArgumentException"
                        },
34                  { SWIG_JavaNullPointerException, "java/lang/NullPointerException" },
35                  { SWIG_JavaDirectorPureVirtual, "java/lang/RuntimeException" },
36                  { SWIG_JavaUnknownError, "java/lang/UnknownError" },
37                  { (SWIG_JavaExceptionCodes)0, "java/lang/UnknownError" }
38          };
39          const SWIG_JavaExceptions_t *except_ptr = java_exceptions;
40
41          while (except_ptr->code != code && except_ptr->code)
42                  except_ptr++;
43
44          (*jenv)->ExceptionClear(jenv);
45          excep = (*jenv)->FindClass(jenv, except_ptr->java_exception);
46          if (excep)
47                  (*jenv)->ThrowNew(jenv, excep, msg);
48  }
49
50  /* Contract support */
51  #define SWIG_contract_assert(nullreturn, expr, msg) if (!(expr)) {SWIG_JavaThrowException(jenv,
        SWIG_JavaIllegalArgumentException, msg); return nullreturn; } else
52
53  #include "stdio.h"
54
55  SWIGEXPORT void JNICALL Java_helloJNI_hello(JNIEnv *jenv, jclass jcls) {
```

```
56          (void)jenv;
57          (void)jcls;
58          hello();
59  }
```

ソースコード **2.4**　**Java** 用ラッパーファイル

```
1  public class helloJNI {
2          public final static native void hello();
3  }
```

ソースコード **2.5**　**Java** プログラム

```
1  public class hello {
2          public static void hello() {
3                  helloJNI.hello();          // C プログラムの hello()を実行
4          }
5  }
```

2.2　ネットワークプログラミング

本研究が適用対象とする通信ライブラリは，UDP (User Datagram Protocol) と TCP (Transmission Control Protocol) 通信用のソケット API (Application Programming Interface) を提供している．そのため，本論文では UDP を例に挙げて，UDP ネットワークプログラミングに関してとそれを実現するために C 言語や Java，Node.js が標準で提供している UDP 通信用 API ついて述べる．

2.2.1　UDP ネットワークプログラミング

UDP を用いたネットワークプログラムを作成する際に必要となる処理と実装は，送信側と受信側で異なる．

送信側では，特定の IP アドレスと UDP ポート番号で待機している受信側に対して，パケットを送信するという処理が必要となる．そのため，以下の 2 つの実装が必要である．

- ソケットの生成
- 宛先を指定し，データを送信

受信側では，送信側から送られるパケットを特定の UDP ポート番号で待機し，受信するという処理が必要となる．そのため，以下の 3 つの実装が必要である．

- ソケットの生成
- ソケットへの IP アドレスとポート番号の設定
- データの受信

ソケットへの IP アドレスとポート番号の設定は bind と呼ばれる．上記より，UDP ネットワークプログラムを作成する際には最低でもソケット生成，bind，送信，受信の 4 つの機能を持つ API を使用する必要がある．

2.2.2 C 言語

C 言語で標準提供されている UDP 通信用 API を表 1 に示す．また，これらを用いた UDP 通信プログラムの送信側をソースコード 2.6，受信側をソースコード 2.7 に示す．

送信側では，最初に 6 行目でソケットを生成する．この際，第 1 引数でプロトコルファミリー，第 2 引数で通信方式，第 3 引数でソケットによって使用される固有のプロトコルを指定する．サンプルプログラムの場合は，プロトコルファミリーに IPv4，通信方式に UDP，使用される固有のプロトコルにプロトコルファミリーの既定のプロトコルを指定し，ソケットを生成した．その後，8〜10 行目で送信先のアドレス情報を設定する．最後に，12 行目で受信側にデータを送信する．

受信側では，最初は送信側と同様に 7 行目でソケットを生成し，9〜11 行目で受信相手に関するアドレス情報を設定する．次に 13 行目で `bind()` をすることで生成したソケットと設定したアドレス情報を紐付ける．その後，16 行目で送信側からデータを受信する．

表 1　C 言語で提供されている API

処理	API
ソケット生成	**int socket**(**int** *domain*, **int** *type*, **int** *protocol*);
bind	**int bind**(**int** *sockfd*, **const struct sockaddr** **addr*, **socklen_t** *addrlen*);
送信	**ssize_t sendto**(**int** *sockfd*, **const void** **buf*, **size_t** *len*, **int** *flags*, **const struct sockaddr** **dest_addr*, **socklen_t** *addrlen*);
受信	**ssize_t recvfrom**(**int** *sockfd*, **void** **buf*, **size_t** *len*, **int** *flags*, **struct sockaddr** **src_addr*, **socklen_t** **addrlen*);

ソースコード 2.6　C 言語の送信側サンプルプログラム

```
1  int main() {
2          /* 変数の宣言 */
3          int sock;           // ソケット
4          struct sockaddr_in addr;
5
6          /* ソケット生成 */
7          if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) > 0) {
8                  exit(1);
9          }
10
11         /* 接続先のIP アドレスとポート番号を指定 */
12         addr.sin_family = AF_INET;
13         addr.sin_port = htons(4330);
14         addr.sin_addr.s_addr = inet_addr("127.0.0.1");
15
16         sendto(sock, "C", 1, 0, (struct sockaddr *)&addr, sizeof(addr));          // 送信
17
18         close(sock);          // ソケットを閉じる
19
20         return 0;
21 }
```

ソースコード 2.7　C 言語の受信側サンプルプログラム

```
1  int main() {
2          /* 変数の宣言 */
3          int sock;              // ソケット
4          struct sockaddr_in addr;
5          char buf[1024];
6
7          /* ソケット生成 */
8          if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) > 0) {
9                  exit(1);
10         }
11
12         /* IP アドレスとポート番号を指定 */
13         addr.sin_family = AF_INET;
14         addr.sin_port = htons(4330);
15         addr.sin_addr.s_addr = INADDR_ANY;
16
17         bind(sock, (struct sockaddr *)&addr, sizeof(addr));          // bind
18
19         memset(buf, 0, sizeof(buf));
20         recvfrom(sock, buf, sizeof(buf), 0, NULL, NULL);          // 受信
21
22         close(sock);          // ソケットを閉じる
23
24         return 0;
25 }
```

2.2.3　Java

Java[*2]で標準提供されている UDP 通信用 API を表 2 に示す．また，これらを用いた UDP 通信プログラムの送信側をソースコード 2.8，受信側をソースコード 2.9 に示す．

送信側では，7 行目で UDP 通信用ソケットを生成する．この際，Java の仕様によって，IPv6 アドレスが使用可能な場合は IPv4 アドレスよりも IPv6 アドレスを優先し，自動的に使用する．その後，送信するデータを 13 行目で生成する．最後に 15 行目でデータを送信する．

受信側では，8 行目でソケット生成メソッドに受信待機するポート番号を引数として渡すことでソケット生成と `bind()` を同時に実行する．その後，12 行目で送信側からのデータを受信する．

ソースコード 2.8　Java の送信側サンプルプログラム

```
1  public static void main(String[] args) {
2          /* 変数の宣言 */
3          DatagramSocket ds = null;          // ソケット
4          InetSocketAddress isa = new InetSocketAddress("127.0.0.1", 4330);          // 接続先の
               IP アドレスとポート番号を指定
5
```

[*2]https://java.com/

表 2　**Java** で提供されている **API**

処理	API
ソケット生成	DatagramSocket **DatagramSocket**()
bind	void **DatagramSocket.bind**(**SocketAddress** *addr*)
送信	void **DatagramSocket.send**(**DatagramPacket** *p*)
受信	void **DatagramSocket.receive**(**DatagramPacket** *p*)

```
6           try {
7                   ds = new DatagramSocket();          // ソケット生成
8
9                   String message = "Java";
10                  byte[] buf = new byte[1024];
11                  buf = message.getBytes("UTF-8");
12
13                  DatagramPacket packet = new DatagramPacket(buf, buf.length, isa);
14
15                  ds.send(packet);            // 送信
16          } catch (IOException e) {
17                  e.printStackTrace();
18          } finally {
19                  if (ds != null) {
20                          ds.close();             // ソケットを閉じる
21                  }
22          }
23  }
```

ソースコード **2.9**　**Java** の受信側サンプルプログラム

```
1   public static void main(String[] args) {
2           /* 変数の宣言 */
3           DatagramSocket ds = null;           // ソケット
4
5           try {
6                   byte[] buf = new byte[1024];
7
8                   ds = new DatagramSocket();          // ソケット生成
9
10                  ds.bind(new InetSocketAddress(4330));           // bind
11
12                  DatagramPacket packet = new DatagramPacket(buf, buf.length);
13
14                  ds.receive(packet);             // 受信
15          } catch (IOException e) {
16                  e.printStackTrace();
17          } finally {
18                  if (ds != null) {
```

```
19                              ds.close();             // ソケットを閉じる
20                      }
21              }
22      }
```

2.2.4 Node.js

Node.js[*3]で標準提供されている UDP 通信用 API を表 3 に示す．また，これらを用いた UDP 通信プログラムの送信側をソースコード 2.10，受信側をソースコード 2.11 に示す．

送信側では，1 行目で UDP 通信用のソケットを IPv4 または IPv6 を指定して生成する．そして 6 行目で相手の情報を引数で与えてデータを送信する．

受信側では，送信側と同様に 1 行目でソケットを生成する．その後 9 行目で受信サーバに対して受信するデータの種類を設定する．そして，12 行目でサーバが受信に使用するソケットにポート番号と自身のアドレス情報を紐付ける `bind()` を行う．

表 3　Node.js で提供されている API

処理	API
ソケット生成	**dgram.createSocket**(**string** *type*)
bind	**dgram.createSocket.bind**(**options**{**integer** *port*, **string** *address*, **boolean** *exclusive*, **integer** *fd*})
送信	**dgram.createSocket.send**({**Buffer** | **Uint8Array** | **string** | **Array**} *msg*, **integer** *port*)
受信	**dgram.createSocket.on**(‘*message*’, (**Buffer** *msg*, **Object** *rinfo*) =>{ })

ソースコード 2.10　Node.js の送信側サンプルプログラム

```
1  const client = dgram.createSocket('udp4');          // ソケット生成
2
3  const message = new Buffer('Node.js');
4
5  /* IP アドレスとポート番号を指定して，送信 */
6  client.send(message, 0, message.length, 4330, '127.0.0.1', function(err, bytes) {
7          if (err) throw err;
8          client.close();          // ソケットを閉じる
9  });
```

ソースコード 2.11　Node.js の受信側サンプルプログラム

```
1  const server = dgram.createSocket('udp4');          // ソケット生成
2
3  server.on('listening', function() {
4          const address = server.address();
5  });
6
7  /* 受信 */
```

[*3]https://nodejs.org/

```
8  server.on('message', function(message, remote) {
9          console.timeEnd(time);
10 });
11
12 server.bind(4330, '127.0.0.1');          // bind
```

2.2.5 言語による特徴

ソケット生成時には IPv4 または IPv6 といったプロトコルファミリーの指定と TCP や UDP 通信方式の指定が必要となる．C 言語では，プロトコルファミリーと通信方式をソケット生成関数に引数で細かく渡す必要がある．それに対して Java では，プロトコルファミリーは Java の仕様によって自動的に IPv4 または IPv6 が選択される．そして，通信方式はソケットのインスタンスを生成する際のクラスによって決められる．Node.js では，通信方式はソケット生成時のクラスによって決められるが，プロトコルファミリーはソケット生成時の引数で渡す必要がある．

bind 時には，生成したソケットに対して IP アドレスとポート番号を設定する必要がある．C 言語では，データを送信する前に構造体にプロトコルファミリーとポート番号，IP アドレスを格納し，データ送信時にこの構造体を引数に渡す必要がある．Java でも同様に，送信するデータを作成する前にアドレス情報を格納するインスタンスを生成し，そこに IP アドレスとポート番号を格納する．その後，データ生成時にこのアドレス情報を格納したインスタンスを引数に渡すことで実現している．Java では受信側に限り，ソケットのインスタンス生成時にポート番号を引数に渡すことでソケット生成と bind を同時に行うことも可能である．Node.js では，データ送信時は引数にポート番号と IP アドレスを渡すことで，受信時はソケット生成後にソケットに対して bind を実行してポート番号と IP アドレスを引数に渡す．そのため，Node.js ではデータ送信前に予め構造体やインスタンスにポート番号や IP アドレスといった情報を格納するといった段階を踏むのではなく，送信時や bind 関数の引数で直接情報を渡している．

以上のことから，プログラミング言語によって IPv4 または IPv6 といったプロトコルファミリーを引数で渡す必要がある場合と自動的に選択される場合といった違いがある．また，TCP や UDP といった通信方式についてもソケット生成時に引数で選択する場合とソケット生成時に用いるクラスによって予め通信方式を選択する場合といった違いがある．そして，bind 時にもアドレス情報を予め作成してデータ送信時に引数で与える場合とデータ生成時に引数で与える場合，予めアドレス情報を作成せずにデータ送信時に直接引数に渡す場合といった違いがある．従って，高級言語から低級言語の機能を呼び出す場合は，高級言語の仕様に合わせて上記のような違いをプログラム内で除去することで，高級言語と同じ使用方法で低級言語の機能を使用可能にする必要がある．

第3章　提案方式

本章では，ラッパーもしくはVPNを用いることで高級言語の標準通信機能を機能拡張したC言語通信ライブラリをアプリケーションに適用する2つの方式について述べる．

3.1　目的

既存のC言語ライブラリを他の高級言語から使用する場合，ラッパーを用いて使用するC言語ライブラリの使用方法で高級言語を使用したアプリケーションを開発する必要がある．

この際，一般的に標準で高級言語に存在しない新機能の場合であれば問題はない．しかし，通信機能といった一般的に標準で高級言語に備わっている機能を拡張した機能を使用したい場合は，これらの拡張機能は高級言語と同じ使用方法であることが望ましい．

また，C言語ライブラリによる通信を既存のスマートフォンアプリケーションに提供したい場合は，上記のラッパーを用いた方式ではアプリケーションを改造する必要があるため使用できない．そのため，Android OSとiOSによって提供されているVPNの機能を利用することで，スマートフォンユーザに通信方式を提供することが可能である．

そこで本研究では，ラッパーを用いて通信ライブラリ機能を高級言語の開発者に提供するラッパー方式とVPNを用いて通信ライブラリ機能を直接ユーザのスマートフォンに提供するVPN方式の2つの方式を提案する．ラッパー方式では，C言語やC++ライブラリの機能をラップすることで，高級言語と同じ使用方法でC言語やC++の機能を機能拡張された通信を実現することを目的とする．VPN方式では，アプリケーションに一切変更を加えることなく，機能拡張された通信を実現することを目的とする．

3.2　ラッパー方式

3.2.1　ラッパー技術

ラッパーとは，あるプログラミング言語から他のプログラミング言語で実装された関数等の機能を利用するための仕組みを利用するFFI (Foreign Function Interface) という仕組みを用いて，高級言語からC言語やC++，OS固有の機能を使用するための技術である．ラッパーはFFIの仕組みを用いることで，C言語やC++のライブラリ (以下，Cライブラリと呼ぶ．) を一切変更することなく，高級言語から使用することが可能である．しかし，ラッパーは高級言語のプログラミング言語で実装する必要がある．この際，ラッパーではCライブラリの関数と同一名かつ同一引数，同

一型で定義しなくてはならない．つまり，ラッパーを使用する高級言語では，C ライブラリを意識したプログラミングが必要である．アプリケーション開発者は 2 つの言語に跨った知識が必要となり，開発の負荷が増大する．

ラッパーはラッパー自身をラップするといった，2 重ラップが可能である．そのため，1 回目のラップで FFI の仕組みを用いて高級言語から C 言語や C++の機能を使用可能にするラッパーを作成する．次に，2 回目のラップによって 1 回目のラッパーで使用可能になった機能を高級言語と同じ使用方法にするラッパーを作成する．これにより，アプリケーション開発者は高級言語のみを意識し，開発が可能になる．

3.2.2 ラッパー方式の動作

図 2 にラッパー方式の動作を示す．C ライブラリの機能をラップして高級言語から使用可能するラッパーを FFI ラッパーとし，FFI ラッパーをラップして C ライブラリの機能を高級言語と同じ使用方法で使用できるようにするラッパーを API ラッパーとする．

FFI ラッパーでは，FFI の仕組みを用いて C ライブラリが提供する API を呼び出し元の高級言語から使用可能にする．この際，ラッパーでは C ライブラリと呼び出し元の高級言語での関数や引数の型のマッピングを行う．型のマッピングを行うことで，高級言語から C ライブラリの API を呼び出し，C ライブラリと同じ使い方をする場合に限り API を使用することが可能になる．また，C ライブラリ独自の API は FFI ラッパーから使用可能である．

API ラッパーでは，FFI ラッパーによって使用可能になった C ライブラリの API を高級言語の使用方法と同じ使用方法にする．呼び出し元の高級言語の通信用クラスを継承し，継承したクラスに C ライブラリの API を実装する．これにより，この通信用クラスの使用方法は継承元のクラスと同じ使用方法，すなわち高級言語の標準ライブラリと同じ使用方法になる．継承したクラスに API を実装する上で高級言語の標準 API の引数では調整することのできないパラメータが存在する場合がある．この場合は，高級言語の仕様に基づいた処理を行うようにパラメータをラッパー内で指定する．

通信機能を持つ高級言語アプリケーションは，高級言語の標準通信 API に代わり，API ラッパーに実装された API を使用することで高級言語と同じ使用方法で，C ライブラリを使用することができる．

3.3 VPN 方式

3.3.1 VPN 技術

VPN とは，パブリックネットワーク上にプライベートネットワークを拡張し，遠隔地から同一ネットワークに接続しているかのように見せる通信を行う技術である．VPN による通信は，全てのパケットに対してカプセル化と暗号化が行われるため安全性が高い．そのため，自宅から職場へリモート接続する場合やある拠点間の通信を行う場合等で使用されている．

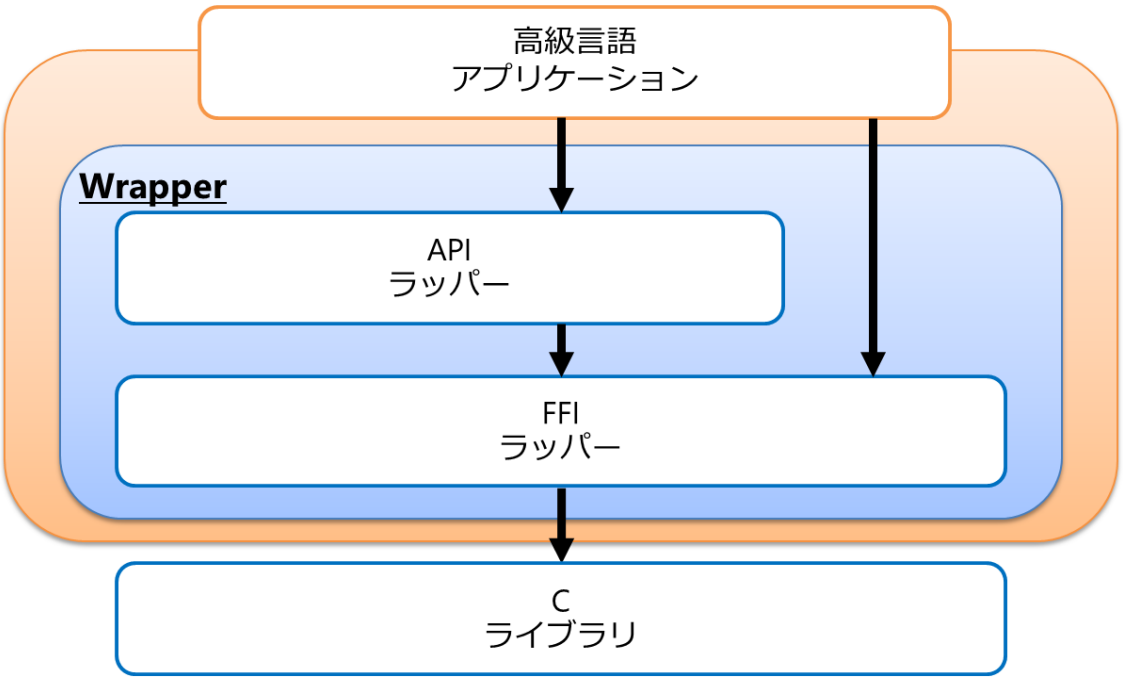

図 2　ラッパー方式の動作

アプリケーションは構築したプライベートネットワーク上のプライベート IP アドレスを使用して通信を行うが，実ネットワークではプライベート IP アドレスを使用することができない．そのため，VPN を用いて通信を行う際は IP ヘッダ等を使用してカプセル化/デカプセル化を行う．

VPN は，TUN (TUNnel) と呼ばれる仮想インタフェースを生成し，プライベート IP アドレスが付与されたパケットのカプセル化/デカプセル化を実現している．この仮想インタフェースの生成や削除は，Linux や Windows，macOS，Android OS，iOS といった OS で実行できるが，root 権限が必要である．Android OS や iOS では一般的に root 権限を取得できないが，Android OS や iOS のアプリケーション開発用に提供された VPN 通信用 API を使用することで，仮想インタフェースの生成やルーティングテーブルの設定，仮想インタフェースが受信したパケットの取得が可能である．また，ユーザはスマートフォンに VPN アプリケーションをインストールするのみで，VPN 通信を使用できる．

VPN アプリケーションの例として，インターネット広告をブロックしたり，モバイルデータ通信量を計測する VPN アプリケーションがある．インターネット広告をブロックする VPN アプリケーションでは，パケットをフックして解析した結果 DNS クエリかつドメイン名が広告配信サイトであればそのパケットの応答に対して偽の結果を返すことでインターネット広告の表示を無くすことが可能である．また，モバイルデータ通信量の計測では，アプリケーションのパケットは全て Virtual I/F を経由するため，VPN アプリケーションでモバイルデータ通信によるパケットのサイズを全て計測し，集計することでモバイルデータ通信量を計測することが可能である．

3.3.2　VPN 方式の動作

VPN 方式では，アプリケーションが生成したパケットをフックし，通信ライブラリに受け渡して処理をさせることにより実現する．図 3 に VPN 方式の動作を示す．

アプリケーションはパケットを生成し送信すると，VPN アプリケーションによって事前に作成した仮想インタフェースがパケットをフックする．パケットをフックした後は，Android OS の場合は Java ラッパー，iOS の場合は Swift ラッパーを経由して，C ライブラリに処理を受け渡す．C ライブラリがパケットに変更を加えた後は，実インタフェースを経由して実ネットワーク上にパケットが送信される．アプリケーションがパケットを受信する際は，上記と逆の動作をする．VPN 方式で使用可能な C ライブラリは，IP/TCP/UDP ヘッダでカプセル化を行う通信ライブラリに限定される．

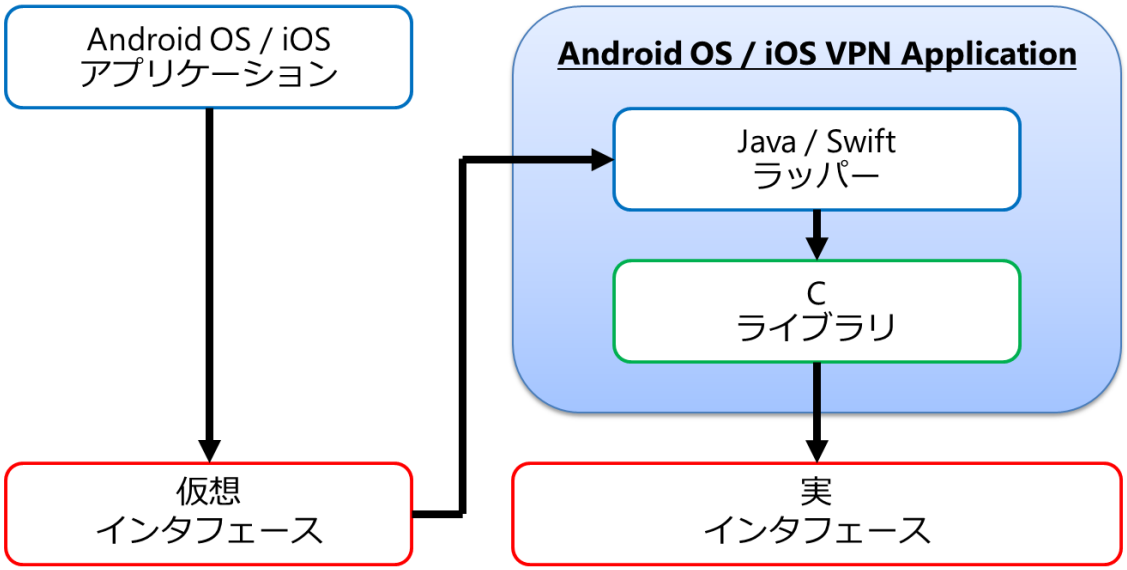

図 3　VPN 方式の動作

第4章　通信ライブラリ (NTMobile framework library)

本章では，提案方式を適用する通信技術である NTMobile (Network Traversal with Mobility) とそれを通信ライブラリ化した NTMfw (NTMobile framework library) の動作と利用モデルについて述べる．

4.1　NTMobile の概要

NTMobile とは，ネットワークが切り替わった際にも通信を継続可能な移動透過性とネットワーク環境に関わらず通信を開始することが可能な通信接続性を同時に実現する技術である．図 4 に NTMobile の概要を示す．NTMobile は下記の機器により構成される．

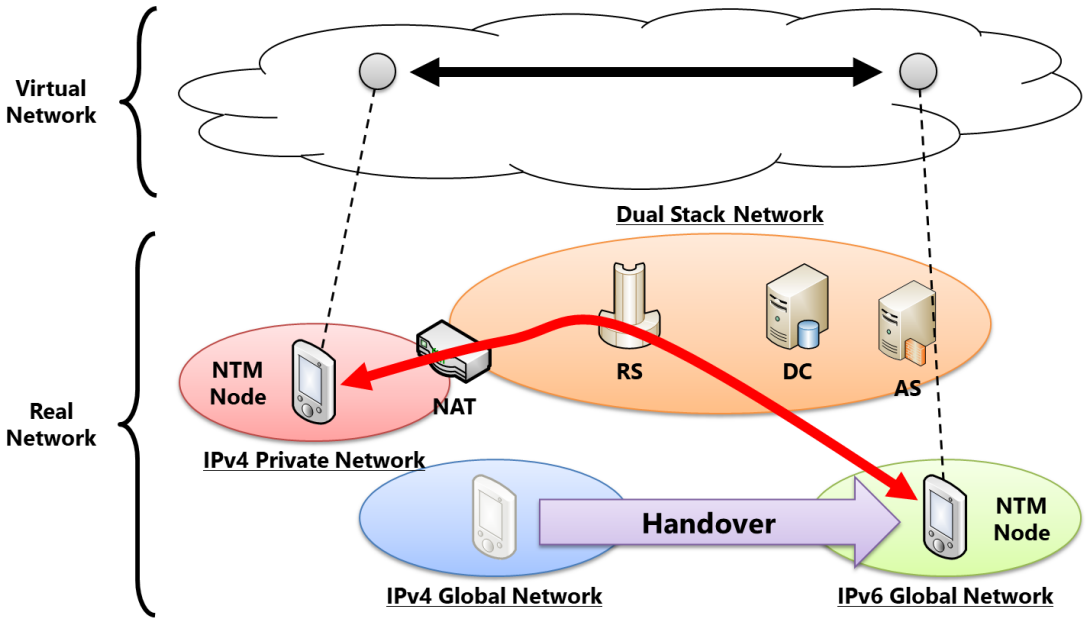

図 4　NTMobile の概要

- NTM 端末 (NTM Node)
  - NTMobile の機能を持つエンド端末．
- AS (Account Server)
  - NTM 端末の認証や DC と NTM 端末間での通信の暗号化に用いられる共通鍵の配布を行う機器．

- DC (Direction Coordinator)
  - NTM 端末の端末情報や仮想 IP アドレスを管理し，トンネル構築の指示を行う機器.
- RS (Relay Server)
  - NTM 端末間でエンドツーエンド通信ができない場合や，一般端末 GN (General Node) と通信する場合にパケットを中継する機器.

NTM 端末は初回起動時に AS にログインし，DC の FQDN (Fully Qualified Domain Name) と DC との通信に用いる共通鍵を取得する．この際，NTMobile を使用するためのアカウントを予め AS に登録しておく必要がある．その後，DC に自身の実 IP アドレス等の情報を登録し，仮想 IPv4/IPv6 アドレスを取得する．これ以降，NTM 端末は DC と定期的に Keep Alive を行うことで，NTM 端末と DC 間で常に通信できる状態を維持する．

通信を開始する際は，NTM 端末 MN (Mobile Node) が実行する，通信相手 NTM 端末 CN (Correspondent Node) の DNS クエリ (Domain Name System query) をトリガとしてトンネル構築処理を開始する．MN は CN の FQDN を用いて，DC に経路指示を要求する．すると DC は，FQDN を用いて CN が NTM 端末か一般端末かどうかを判別する．

CN が NTM 端末であった場合は，MN と CN に対して経路指示を行う．経路指示パケットには，MN と CN 間で UDP トンネルを構築時に用いる共通鍵を生成し，格納する．MN と CN は経路指示パケットに従って MN と CN 間で UDP トンネルを構築する．MN は DNS クエリの応答に CN の仮想 IP アドレスを受け取ることにより，アプリケーションは仮想 IP アドレスを用いた通信を行う．

MN と CN が異なる NAT (Network Address Translation) 配下に存在する場合や IPv4 と IPv6 での通信といった直接通信できない場合は，DC が RS に対して中継指示を行う．そのため，MN と CN は RS 経由の UDP トンネルを構築する．

CN が一般端末であった場合は，MN と RS が UDP トンネルを構築して RS は NAT 処理をすることで，CN との通信を中継する．

4.2 NTMfw の動作

NTMfw は NTMobile の機能をアプリケーションライブラリ化した C 言語通信ライブラリである．図 5 に NTMfw のモジュール構成を示す.

- BSD ソケット API
  - NTMfw がパケットを送受信する際に用いる C 言語の標準ソケット API で，制御メッセージやカプセル化パケットは全てこの API で送受信される.
- NTM ソケット API
  - BSD ソケット API に代わってアプリケーションに提供するソケット API で，NTMfw を初期化する API と名前解決を担う API を除き，仮想 TCP/IP スタックに処理を渡す.
  - NTMfw を初期化する API は，引数から得た AS の FQDN と NTMobile のアカウント情報で AS にログイン後，DC に自身の実 IP アドレス等を登録して仮想 IP アドレスを取得

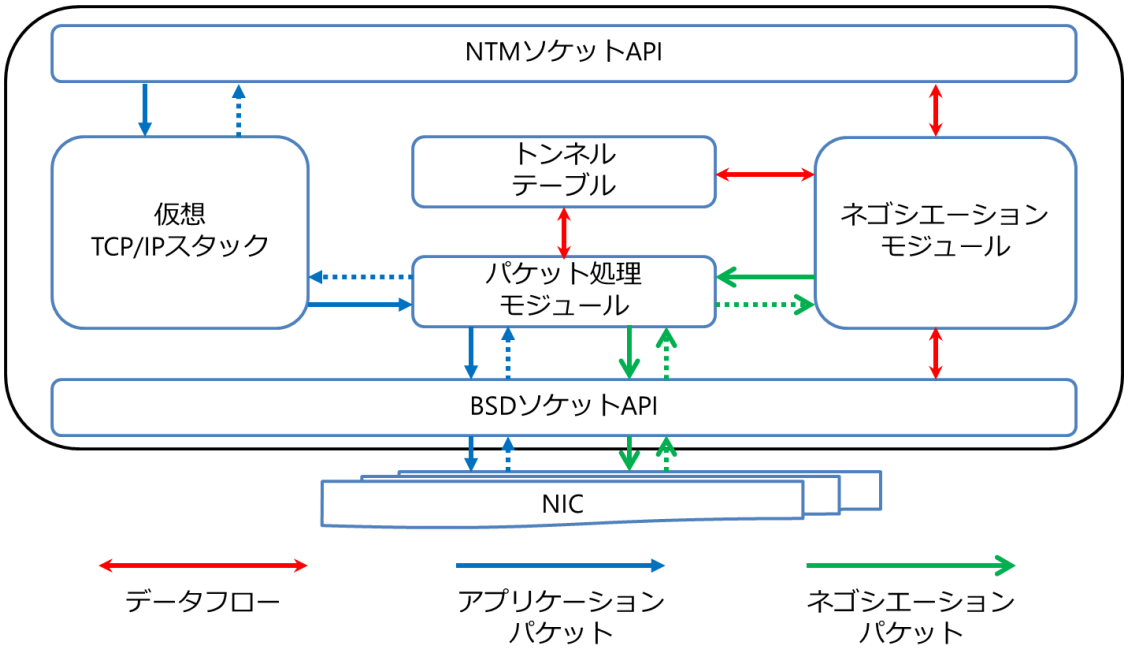

図 5　NTMfw のモジュール構成

する．仮想 IP アドレス取得後，DC と定期的な Keep Alive を開始することで，NTMfw の初期化が完了する．

- 名前解決を担う API は，引数から FQDN を抽出してトンネル構築を行い，当該 FQDN に対応する仮想 IP アドレスを返す．仮想 IP アドレスは，引数に応じて仮想 IPv4/IPv6 アドレスの片方もしくは両方を返す．

- 仮想 TCP/IP スタック
  - アプリケーションが送受信するデータの TCP/IP 処理を行う．TCP/IP スタックとして lwIP (A Lightweight TCP/IP stack)[*4] を用いている．アプリケーションが送信するデータは，lwIP によって仮想 IP ヘッダが付与され，アウトプットコールバック関数によってパケット処理モジュールに処理が移る．また，パケット処理モジュールから受信したパケットは，lwIP のインプット関数に処理が渡される．
- パケット処理モジュール
  - パケットの MAC (Message Authentication Code) 付与/検証及び暗号化/復号を行い，パケットの種類に応じてネゴシエーションモジュールと仮想 IP スタックとに処理を振り分ける．また，BSD Socket API を用いたパケットの送受信を行う．
- ネゴシエーションモジュール
  - NTMobile の制御メッセージの処理や，アドレス情報の監視を行う．NTM ソケット API の名前解決を担う関数が呼び出された場合や，他端末から通信要求があった場合，このモジュールでトンネル構築処理が行われ，トンネルテーブルが更新される．また，端末の IP アドレスを毎秒確認し，IPv4 アドレスまたは IPv6 アドレスに変化があった場合，

[*4] https://savannah.nongnu.org/projects/lwip/

DC に対してアドレス情報の更新を行い，構築されている全てのトンネルを再構築する．

- トンネルテーブル
  - 通信相手ごとに FQDN，仮想 IPv4/IPv6 アドレス，実 IPv4/IPv6 アドレス，共通鍵，Path ID，Node ID，RS の実 IPv4/IPv6 アドレス等をメンバとするエントリ持つ．複数のキーを持つハッシュテーブルで実装され，ハッシュキーは FQDN，仮想 IPv4/IPv6 アドレス，Path ID，Node ID である．一定時間参照されなかったエントリは，自動的に削除される．

図 6 に NTMfw の動作を示す．NTMfw は仮想 TCP/IP スタックの機能を有しており，この仮想 TCP/IP スタックを持つ仮想ネットワークインタフェースに仮想 IP アドレスが割り当てられる．NTMfw は OS 標準のソケット API を使用してパケットの送受信を行う．アプリケーションは NTM ソケット API を使用してデータの送受信を行う．アプリケーションが送信したデータは，仮想 TCP/IP スタックの処理により，仮想 IP アドレスを用いて TCP または UDP ヘッダと IP ヘッダが付与される．このパケットは暗号化や MAC 付与等の処理後，BSD ソケット API を用いて UDP で送信される．この処理により，NTM ソケット API で送信するパケット UDP でカプセル化されて送信される．カプセル化されたパケットを NTMfw が BSD ソケット API で受信すると，外部の UDP/IP ヘッダが取り除かれる．受信したパケットの MAC 検証と復号を行い，仮想 TCP/IP スタックに渡すことで内部の IP ヘッダと TCP または UDP ヘッダが除去されることで，アプリケーションはデータを受信できる．

NTMfw は初期化時される際に仮想ネットワークインタフェースに仮想 IPv4/IPv6 アドレスを設定する．また，端末が属する IPv4 または IPv6，IPv4/IPv6 デュアルスタックネットワークといったネットワーク環境に応じて，IPv4 ソケットまたは IPv6 ソケット，IPv4/IPv6 の両方のソケットで通信可能な状態を維持する．NTMfw は NIC (Network Interface Card) に割り当てられた実 IP アドレスを監視しており，この実 IP アドレスの変化を移動と認識する．移動を検知した場合，IPv4/IPv6 ソケットを再生成して，新たな実 IP アドレスの登録と UDP トンネルの再構築を行う．この際，仮想 IP アドレスは変化しないため，実 IP アドレスの変化をアプリケーションに隠蔽する．

これらの処理によって，アプリケーションは NTM ソケット API を使用することで仮想 IP アドレスによるパケットの送受信が可能となり，実 IP アドレスに依存することなく通信を行うことができる．

4.3 NTMfw の利用モデル

NTMfw の利用モデルにはフレームワーク組込型と TUN 利用型 [10] が存在する．図 7 にフレームワーク組込型の利用モデル，図 8 に TUN 利用型の利用モデル，表 4 に NTMfw の各利用モデルの特徴を示す．

フレームワーク組込型は，アプリケーション開発時にプログラミング言語の標準ソケット API を用いる代わりに，NTMfw の提供する NTM ソケット API を用いる利用モデルである．動作する OS は Linux のみに限らず，Windows や macOS，Android OS，iOS にも適用することが可能であ

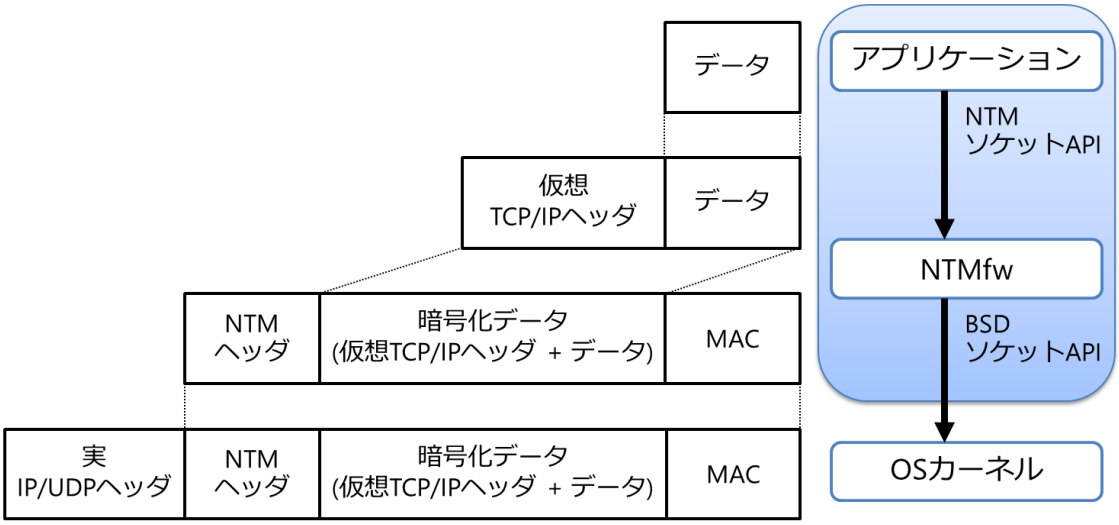

図 6　NTMfw の動作

り，管理者権限 (root 権限) が必要ない．しかし，使用するにはアプリケーション毎に NTMfw を組み込むための改造が必要となる．

TUN 利用型は，一般アプリケーションによって生成されたパケットをネットワーク上に送信する前にフックして別アプリケーションに渡すという TUN インタフェースの仕組みを用いて NTMobile 通信を実現する利用モデルである．Linux や Windows，macOS でも利用することが可能だが，TUN デバイスの生成には root 権限が必要である．しかし，アプリケーションを一切改造する必要がなく，複数のアプリケーションで使用可能である．

表 4　NTMfw の利用モデル

	フレームワーク組込型	TUN 利用型
使用方法	NTM ソケット API	TUN インタフェース
動作 OS	Linux/Windows/macOS/Android OS/iOS	Linux/Windows/macOS
アプリケーションの改造	必要	不要
管理者権限	不要	必要

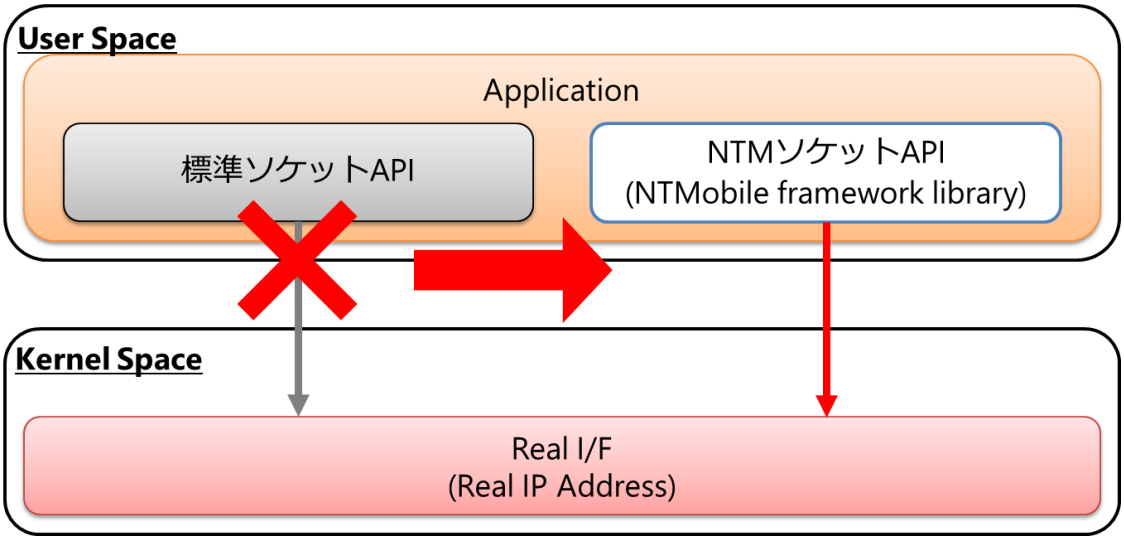

図 7　フレームワーク組込型の利用モデル

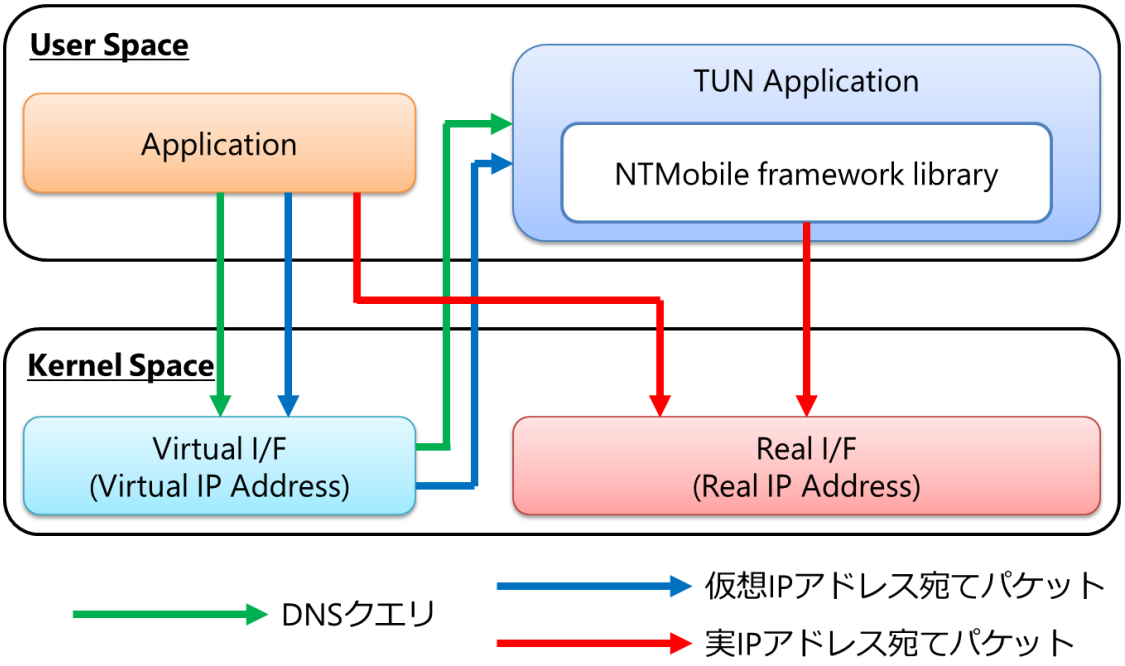

図 8　TUN 利用型の利用モデル

第5章　実装

本章では，C 言語通信ライブラリである NTMfw に対する提案方式の実装について述べる．

5.1　ラッパー方式の実装

4.3 節の NTMfw の利用モデル” フレームワーク組込型” を対象に，Java で NTM ソケット API を使用可能にするラッパーを実装した．図 9 に Java で実装したラッパー方式のモジュール構成を示す．NTMfw 用 Java ラッパーは次のモジュールから構成される．

- NTMobile Socket Class
  - Java の標準ソケット API に代わってアプリケーションに提供するソケット API が実装されたクラスである．アプリケーションはこのクラスを用いて通信することで NTMobile 通信を実現できる．Java の標準 UDP 通信用ソケットクラスである DatagramSocket クラス[⋆5]と DatagramSocketImpl クラス[⋆6]，標準 TCP 通信用ソケットクラスである Socket クラス[⋆7]と SocketImpl クラス[⋆8]，ServerSocket クラス[⋆9]を継承し，各クラスのメソッドをオーバーライドしているため，使用方法は Java の標準ソケットクラスと同じである．
- NTMobile I/O Class
  - Java の標準入出力 API に代わってアプリケーションに提供する入出力 API が実装されたクラスである．アプリケーションは TCP 通信時にこのクラスを用いて送受信を行うことで NTMobile による通信を実現できる．Java の標準バイト入力ストリームのスーパークラスである InputStream クラス[⋆10]と標準バイト出力ストリームのスーパークラスである OutputStream クラス[⋆11]を継承し，各クラスのメソッドをオーバーライドしている．そのため，Java 標準の全バイト入出力クラスに対して使用可能であり，使用方法は Java 標準のバイト入出力クラスと同じである．
- NTMobile API Class
  - FFI の仕組みを用いてマッピングされた NTMfw の API と NTMobile 独自の API が実装されたクラスである．FFI にはオープンソースソフトウェアである JNA (Java Native

[⋆5] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/net/DatagramSocket.html
[⋆6] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/net/DatagramSocketImpl.html
[⋆7] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/net/Socket.html
[⋆8] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/net/SocketImpl.html
[⋆9] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/net/ServerSocket.html
[⋆10] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStream.html
[⋆11] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/OutputStream.html

Access)[*12] を使用した．NTMobile の独自 API には自身の端末情報を登録し，仮想 IP アドレスを取得する API と名前解決を行い通信相手との間でトンネル構築を行う API がある．

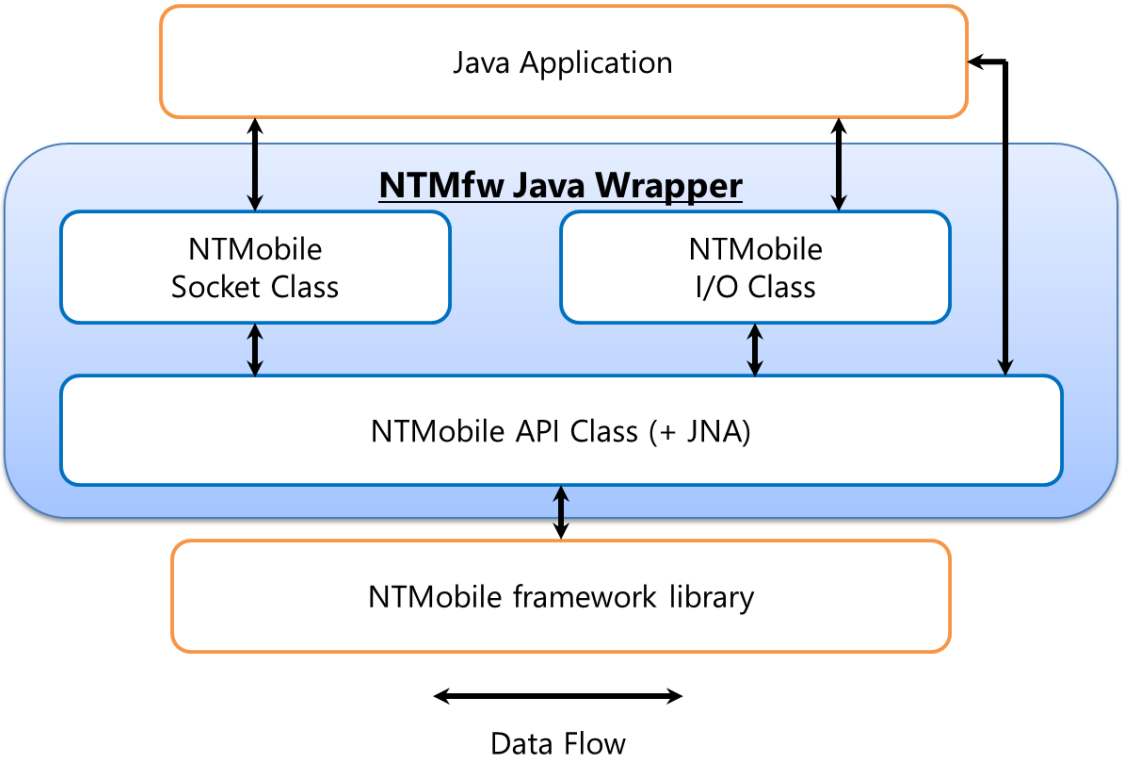

図 9　ラッパー方式のモジュール構成

5.2　VPN 方式の実装

4.3 節の NTMfw の利用モデル”TUN 利用型” を対象に，Swift で NTMfw による通信を iOS アプリケーションに適用する VPN アプリケーションを一部実装した．図 10 に NTMfw を対象に Swift で実装した iOS 用 VPN 方式のモジュール構成を示す．NTMfw 用 iOS VPN アプリケーションは次のモジュールから構成される．

- Virtual I/F (Virtual Interface)
  - iOS アプリケーションのパケットを VPN アプリケーションに中継する仮想インタフェースである．Swift によって提供されているネットワーク拡張クラスである NETunnelProviderManager クラス[*13]を使用して生成する．仮想インタフェースを生成するには，iOS 上に VPN 構成を追加する必要がある．VPN 構成の追加は VPN アプリケーションの初回起動時にのみ行えばよいが，ユーザの許可を得る必要がある．

[*12] https://github.com/java-native-access/jna/
[*13] https://developer.apple.com/documentation/networkextension/netunnelprovidermanager

- Packet Analysis Module
  - NTMfw の機能を利用して，Virtual I/F から受信したパケットの解析を行うモジュールである．パケットを解析した結果，DNS クエリであった場合は Signaling Module に処理を移し，DNS クエリ以外のパケットであった場合は Packet Manipulation Module に処理を移す．
- Signaling Module
  - NTMfw の機能を利用して，VPN アプリケーション起動時の端末情報の登録と Packet Analysis Module から受け取った DNS クエリの処理を行う．DNS クエリの処理では，通信相手の FQDN に応じて処理が変わる．FQDN が NTM 端末のものであった場合は NTMfw の機能を使用して NTMobile のシグナリング処理を行い，トンネル構築を行う．その後，通信相手の仮想 IP アドレスを DNS レスポンスでアプリケーションに返信する．FQDN が一般端末であった場合は DNS クエリをそのまま中継し，DNS サーバに送信する．その後，DNS サーバからのレスポンスをそのまま中継し，アプリケーションに返信する．
- Packet Manipulation Module
  - NTMfw の機能を利用して，DNS クエリ以外のパケットに NTMfw を使用して MAC 認証や暗号化/復号等の処理を行う．NTMfw では仮想 TCP/IP スタックである lwIP を使用してパケットのカプセル化/デカプセル化を行っていたが，VPN 方式では Real I/F を通じて行う．そのため，VPN 方式で使用される NTMfw では仮想 TCP/IP スタックによる処理がスキップされる．

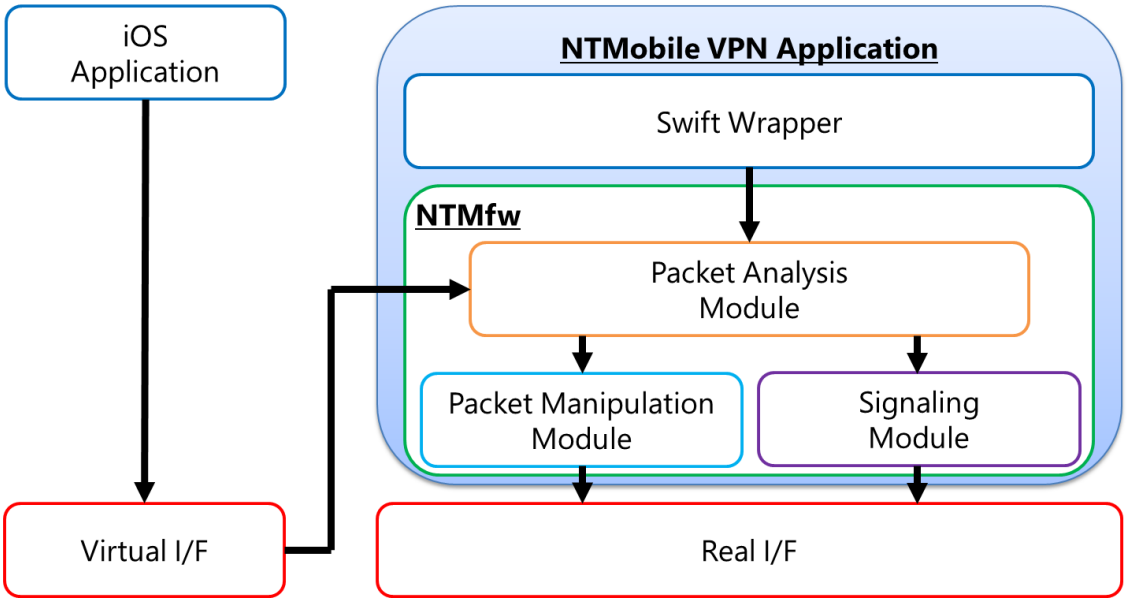

図 10　VPN 方式のモジュール構成

第6章　評価

本章では，5 章で実装したラッパー方式と VPN 方式で使用する TUN 利用型の動作検証及び性能評価について述べる．

6.1　動作検証

実環境にて提案方式が動作することを確認するため，UDP でメッセージを送受信する Java アプリケーションを作成し，動作検証を行った．AS と DC，RS をデュアルスタックネットワーク上に，MN と CN を実機で構築し，MN と CN 間でパケットを送受信した．MN と CN ではラッパー方式と TUN 利用型のいずれかを適用した Java アプリケーションを使用し，4 つ全ての組み合わせで動作検証を行った．図 11 に動作検証を行った実環境のネットワーク構成，デュアルスタックネットワーク上のクラウド機器の仕様を表 5，実機の仕様を表 6，動作検証を行った結果を表 7 に示す．実環境において，ラッパーと TUN 利用型が 4 つ全ての組み合わせで正常に動作することを確認した．

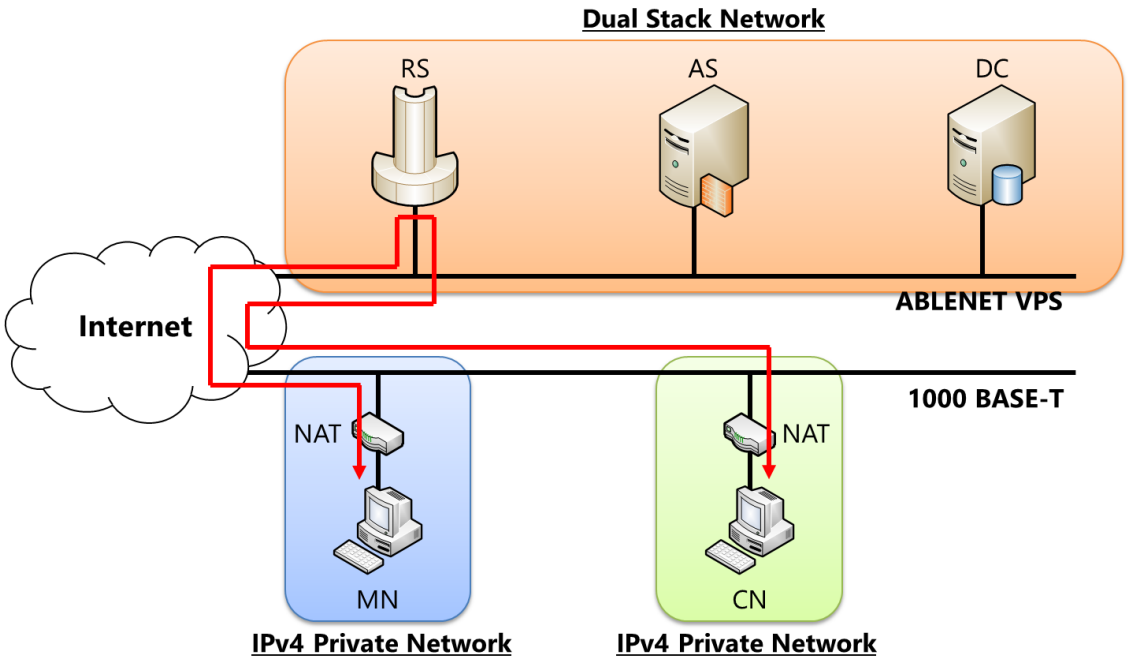

図 11　実環境のネットワーク構成

表 5　クラウド機器の仕様

	AS，DC	RS
OS	CentOS 6 32bit	CentOS 6 32bit
仮想 CPU コア	1Core	2Core
Memory	512MB	1536MB

表 6　実機の仕様

	MN	CN
OS	Ubuntu 14.04 32bit	Ubuntu 14.04 32bit
CPU	Intel Core i7-4770 3.40GHz	Intel Core i7-4770 3.40GHz
Memory	8.00GB	8.00GB

表 7　動作検証の結果

		CN	
		ラッパー	TUN
MN	ラッパー	○	○
	TUN	○	○

6.2　性能評価

ラッパー方式と TUN 利用型における処理時間の差を確認するため，処理時間の計測を行った．ネットワーク環境が処理時間に影響を与えないよう，1 台のホストマシン上に VMware Workstation Player[⋆14]を用いて AS，DC，MN，CN を VM (Virtual Machine) で構築した．図 12 に性能測定を行った仮想環境のネットワーク構成，表 8 にホストマシンの仕様，表 9 に各 VM の仕様を示す．

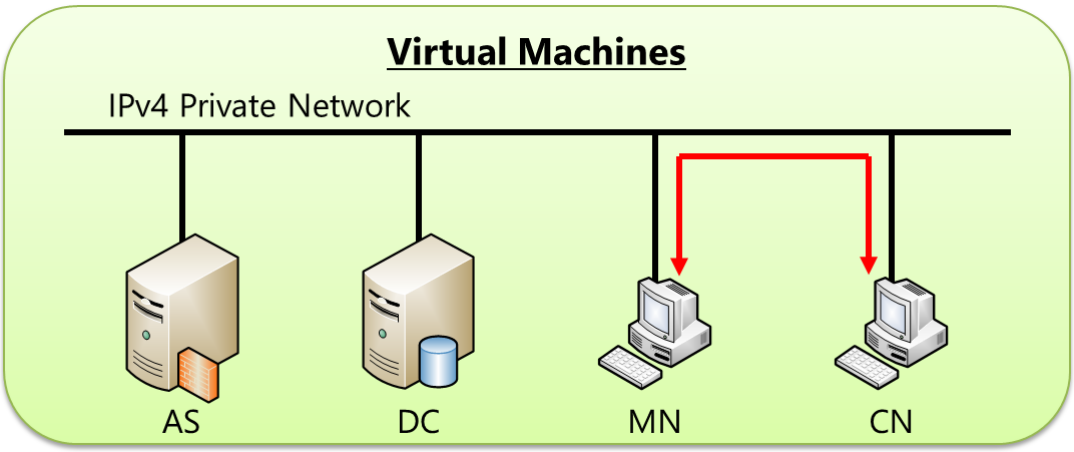

図 12　仮想環境のネットワーク構成

[⋆14]https://www.vmware.com/

表 8　ホストマシンの仕様

	ホストマシン
OS	Windows 10 64bit
CPU	Intel Core i7-4770 3.40GHz
Memory	8.00GB

表 9　各 VM の仕様

	AS，DC	MN，CN
OS	Ubuntu 12.04 32bit	Ubuntu 14.04 32bit
Linux Kernel	3.2.0-101-generic-pae	3.13.0-24-generic
CPU 割り当て	1Core	2Core
Memory 割り当て	1.00GB	2.00GB

AS，DC，MN，CN を同一 IPv4 プライベートネットワークに接続し，MN と CN の両方にラッパー方式を適用した場合と TUN 利用型を適用した場合の 2 通りで処理時間の計測を行った．処理時間の計測には，MN と CN 間で 8 バイト文字列を UDP で 100 回送受信する Java アプリケーションを使用した．送受信されるパケットのサイズは 174[Byte] である．

表 10 にラッパー適用時の平均処理時間，表 11 に TUN 適用時の平均処理時間を示す．ラッパー方式の測定箇所は，ラッパー内 2 箇所と使用する C ライブラリ内 2 箇所の合計 4 箇所である．ラッパー内の測定では，API ラッパーによる API の使用方法を高級言語と同じにする処理と FFI ラッパーによる C ライブラリの機能を実行する処理の 2 箇所で行った．C ライブラリ内の測定では，仮想 TCP/IP スタックである lwIP による TCP/IP 処理と NTMfw によるパケットの MAC 付与/検証，及び暗号化/復号処理の 2 箇所で行った．TUN 利用型の測定箇所は，ラッパー方式の C ライブラリで使用されていた lwIP による TCP/IP 処理は行われないため，NTMfw と TUN の 2 箇所である．

表 10　ラッパー適用時の平均処理時間

測定箇所	内訳	送信時 [μs]	受信時 [μs]
ラッパー	API	10.75	2.08
	FFI	204.49	74.67
C ライブラリ	NTMfw	80.67	162.17
	lwIP	277.95	720.39
合計		573.86	959.31

表 11　TUN 適用時の平均処理時間

測定箇所	送信時 [μs]	受信時 [μs]
NTMfw	51.34	131.68
TUN	0.85	4.90
合計	52.19	136.58

6.3　考察

ラッパー適用時において，表 10 より送信時と受信時における合計処理時間は約 386[μs] の差があった．送受信時の測定箇所をそれぞれ比較すると，lwIP とラッパーにおける処理時間に大きな差があった．lwIP では送信時と受信時に約 442[μs] の差が生じている．受信時には，NTM ソケット API で得たアドレス情報の構造体を lwIP 用のアドレス情報の構造体に変換する処理が行われている．しかし，送信時にはこの処理が不要であるため，この処理が原因となって処理時間に差が生じている．また，ラッパーでも送信時と受信時に約 138[μs] の差が生じている．送信時には，Java 用に生成されたデータを IP アドレスやポート番号などのパラメータ毎に分解し，アドレス長やデータ長を計算し，不足したパラメータを追加する処理が NTMfw の API を使用するために行われている．受信時には，NTMfw によって分解済みのデータを受け取ることが可能であるため，Java 用にデータを変換するのみである．これらの処理の違いが原因となって，処理時間に差が生じている．また，API ラッパーは送信時と受信時ともに，FFI ラッパーの処理時間と比較してごく僅かな時間であった．

TUN 適用時において，表 11 より送信時と受信時における合計処理時間は約 84[μs] とほとんど差がなかった．ラッパー適用時と TUN 適用時とで合計処理時間を比較すると，送信時では約 521[μs]，受信時では約 822[μs] の差が生じている．これには 2 つの原因がある．

1 つ目は，カプセル化/デカプセル化の処理時間である．TUN 適用時の処理時間にはカーネル空間で行われたカプセル化/デカプセル化の処理時間が含まれていない．これに対して，ラッパーでは lwIP を使用してユーザ空間で行われたカプセル化/デカプセル化の処理時間が含まれている．よって，ラッパー適用時の処理時間には TUN に対してカプセル化/デカプセル化の処理時間が余分に含まれている．

2 つ目は，ラッパーという他言語の経由である．ラッパー適用時には C 言語と Java というプログラミング言語間でのデータの違いの除去を行っている．しかし，TUN 適用時にはパケット自体を書き換えているため，プログラミング言語間の違いを除去する必要がない．これにより，ラッパー適用時の処理時間は TUN に対してプログラミング言語間の違いの除去といった処理時間が余分に含まれている．

以上の 2 つの原因が，ラッパー適用時と TUN 適用時における処理時間に差を生じさせている．

6.4 比較

既存方式と提案方式を比較し，2 つの提案方式であるラッパー方式と VPN 方式の 2 つを比較する．

6.4.1 既存方式との比較

SWIG のラッパーと提案方式のラッパー方式を比較した結果を表 12 に示す．

アプリケーション開発時における容易性について，既存方式である SWIG のラッパーでは使用する C 言語または C++ライブラリの使用方法を学ぶ必要があるのに対して，提案方式ではアプリケーション開発時に使用するプログラミング言語の標準 API と同じ使用方法であるため新たに使用方法を学ぶ必要がない．

既存アプリケーションに対する拡張性について，既存方式ではソケット API 自体をすべて書き換える必要があるのに対して，提案方式では API の使用方法が同じであるためクラス名やモジュール名を変更するのみで既存アプリケーションに適用可能である．

処理時間に対する影響力について，提案方式では使用方法を高級言語に合わせるため，既存方式に対してラップが 1 つ多い．そのため，提案方式の方が既存方式よりも処理時間を要する．

既存方式と提案方式のラッパーを比較した結果，提案方式の方がアプリケーションに適用する際の利便性が高いと考える．なぜなら，処理時間のオーバーヘッドについては提案方式の方が劣るが，アプリケーション開発時の容易性や既存アプリケーションに対する拡張性については提案方式の方が既存方式を上回るからである．

表 12　SWIG によるラッパーと提案方式によるラッパーの比較

	SWIG	提案方式
アプリケーション開発時における容易性	×	○
既存アプリケーションに対する拡張性	×	○
処理時間のオーバーヘッド	○	△

6.4.2 提案方式での比較

提案したラッパー方式と VPN 方式を比較した結果を表 13 に示す．

アプリケーション開発時における容易性について，ラッパー方式では使用するクラス名やモジュール名を標準クラスやモジュールとは異なるのに対して，VPN 方式では標準クラスやモジュールを使用して問題ないため従来と変更点は一切ない．

既存アプリケーションに対する拡張性について，クラス名やモジュール名を全て変更する必要があるのに対して，提案方式では一切変更を加えることなく既存アプリケーションに適用可能である．

他の VPN との同時使用については，ラッパー方式では Android OS/iOS の VPN 機能を使用しないため他の VPN との同時使用が可能であるのに対して，VPN 方式では Android OS/iOS の VPN 機能を使用する必要があるため他の VPN アプリケーションとの同時使用が不可能である．

汎用性については，ラッパー方式ではどのような通信ライブラリに対しても適用することが可能だが，VPN 方式は IP/TCP/UDP ヘッダでカプセル化を行う通信ライブラリ以外には適用することが不可能である．

よって，IP/TCP/UDP ヘッダでカプセル化を行う通信ライブラリで且つ他の VPN アプリケーションを使用していない場合は VPN 方式，IP/TCP/UDP ヘッダでカプセル化を行わない通信ライブラリや他の VPN アプリケーションを使用している場合はラッパー方式の方がアプリケーションに適用する際の利便性が高いと考える．

表 13　ラッパー方式と VPN 方式の比較

	ラッパー方式	VPN 方式
アプリケーション開発時における容易性	△	○
既存アプリケーションに対する拡張性	×	○
他の VPN との同時使用	○	×
汎用性	○	×

第7章　結論

本論文では，アプリケーションから機能拡張された通信ライブラリを利用する方法について提案と実装を行った．ラッパーを用いた方式では，C 言語や C++といった低級言語で実装された通信ライブラリを高級言語から使用する際に，高級言語と同じ使用方法によって通信ライブラリを使用可能にした．そして，VPN を用いた方式では，スマートフォン上のアプリケーションが生成するパケットを Virtual I/F 経由させることで，アプリケーションに一切変更を加えることなく通信ライブラリを適用する方法を提案した．

Linux 上で Java 用ラッパーと VPN 方式の土台となる TUN 利用型の動作検証と性能評価を行った．仮想環境で動作検証を行い，提案方式が動作することを確認した．また，提案したラッパー方式と VPN 方式を比較した結果，IP/TCP/UDP ヘッダでカプセル化を行う通信ライブラリで且つ他の VPN アプリケーションを使用していない場合は VPN 方式，IP/TCP/UDP ヘッダでカプセル化を行わない通信ライブラリや他の VPN アプリケーションを使用している場合はラッパー方式の方がアプリケーションに適用する際の利便性が高いと考える．

謝辞

本研究を進めるにあたり，多大なるご指導とご教授を賜りました，指導教官である名城大学大学院理工学研究科　渡邊晃教授に心から感謝致します.

本論文を執筆するにあたり，快く副査を引き受けて頂きました，名城大学大学院理工学研究科　山本修身教授に心より厚く御礼申し上げます.

本研究を進めるにあたり，副査を快諾して頂き，様々なご指導を賜りました，名城大学大学院理工学研究科　鈴木秀和准教授に深く感謝致します.

本論文を執筆するにあたり，快く副査を引き受けて頂きました，名城大学大学院理工学研究科　旭健作准教授に心から厚く感謝致します.

本研究を進めるにあたり，ご意見並びにご助言を賜りました，愛知工業大学大学院経営情報科学研究科　内藤克浩准教授に深く感謝致します.

最後に，本研究を進めるにあたり，数々の有益なご助言を賜りました，渡邊研究室及び鈴木研究室の諸氏に深く感謝致します.

参考文献

[1] Beazley, D. M. and Dumont, D.: Perl Extension Building with SWIG (1998).

[2] M. Beazley, D.: SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++, Vol. 4 (1996).

[3] Beazley, D. and Lomdahl, P.: Feeding a large-scale physics application to Python, *6. international python conference* (1997).

[4] Naito, K., Kamienoo, K., Suzuki, H., Watanabe, A., Mori, K. and Kobayashi, K.: End-to-end IP mobility platform in application layer for iOS and Android OS, *Proc. of IEEE CCNC* (2014).

[5] 納堂博史，八里栄輔，鈴木秀和，内藤克浩，渡邊　晃：実用化に向けた NTMobile フレームワークの実装と評価，研究報告モバイルコンピューティングとパーベイシブシステム（MBL），Vol. 46 (2017).

[6] 納堂博史，鈴木秀和，内藤克浩，渡邊　晃：エンドツーエンド通信をアプリケーションレベルで可能にする通信ライブラリの実現と評価，情報処理学会論文誌，Vol. 60, pp. 16–26 (2019).

[7] 鈴木秀和，上醉尾一真，水谷智大，西尾拓也，内藤克浩，渡邊　晃：NTMobile における通信接続性の確立手法と実装，情報処理学会論文誌，Vol. 54, No. 1, pp. 367–379 (2013).

[8] 内藤克浩，上醉尾一真，西尾拓也，水谷智大，鈴木秀和，渡邊　晃，森香津夫，小林英雄：NTMobile における移動透過性の実現と実装，情報処理学会論文誌，Vol. 54, No. 1, pp. 380–397 (2013).

[9] 上醉尾一真，鈴木秀和，内藤克浩，渡邊　晃：IPv4/IPv6 混在環境で移動透過性を実現する NTMobile の実装と評価，情報処理学会論文誌， Vol. 54, pp. 2288–2299 (2013).

[10] 鈴木秀和，内藤克浩，渡邊　晃：ユーザ空間における移動透過通信技術の設計と実装，マルチメディア、分散協調とモバイルシンポジウム 2014 論文集，pp. 1319–1325 (2014).

研究業績

国際会議（査読あり）

（1）K. Shimizu，H. Suzuki，K. Naito and A. Watanabe: Realization and Evaluation of Java Wrapper that calls the End-to-End Communication Library，*Proc. of The 10th International Conference on Mobile Computing and Ubiquitous Networking (ICMU2017)*，pp.69-70，Oct.2017.

国内会議・研究会・大会等

（1）清水一輝，納堂博史，鈴木秀和，内藤克浩，渡邊晃 : NTMobile で SIP 通信を可能とする仮想 IPv4 アドレス生成方式の検討，平成 28 年度電気関係学会東海支部連合大会論文集，Vol. 2016，講演番号 B2-5，Sep.2016.

（2）清水一輝，八里栄輔，納堂博史，鈴木秀和，内藤克浩，渡邊晃：NTMobile フレームワークの Java ラッパーの提案と実装，第 79 回情報処理学会全国大会講演論文集，Vol. 2017，講演番号 6U-03，pp.369-370，Mar.2017.

（3）清水一輝，納堂博史，鈴木秀和，内藤克浩，渡邊晃：C 言語通信ライブラリを呼び出す Java ラッパーの実現と評価，マルチメディア，分散，協調とモバイル (DICOMO2017) シンポジウム論文集，Vol. 2017，講演番号 2F-3，pp.409-415，Jun.2017.

（4）黒宮魁人，清水一輝，鈴木秀和，内藤克浩，渡邊晃：スマートデバイスにおける NTMobile の経路生成方式の提案，平成 29 年度電気関係学会東海支部連合大会論文集，Vol. 2017，講演番号 C3-5，Sep.2017.

（5）清水一輝，鈴木秀和，内藤克浩，渡邊晃：エンドツーエンド通信ライブラリを使用可能にする Java ラッパーの実現と評価，第 15 回情報学ワークショップ WiNF2017 論文集，Vol. 2017，講演番号 PC-30，Dec.2017.

（6）黒宮魁人，清水一輝，鈴木秀和，内藤克浩，渡邊晃：スマートデバイスにおける NTMobile の経路生成方式の提案，第 15 回情報学ワークショップ WiNF2017 論文集，Vol. 2017，講演番号 D-4，Dec.2017.

（7）清水一輝，鈴木秀和，内藤克浩，渡邊晃：言語の違いを意識することなく C 言語通信ライブラリを利用可能とするラッパーの提案と実装，第 80 回情報処理学会全国大会講演論文集，Vol. 2018，講演番号 6T-06，pp.217-218，Mar.2018.

（8）黒宮魁人，清水一輝，鈴木秀和，内藤克浩，渡邊晃：VPNService を利用した移動透過性の実現方式の提案，第 80 回情報処理学会全国大会講演論文集，Vol. 2018，講演番号 6T-07，

pp.219-220，Mar.2018.
（9）清水一輝，鈴木秀和，内藤克浩，渡邊晃：機能拡張した通信ライブラリを呼び出す Node.js ラッパーの検討，平成 30 年度電気関係学会東海支部連合大会論文集，Vol. 2018，講演番号 H2-5，Sep.2018.
（10）渡邉憲士，清水一輝，鈴木秀和，内藤克浩，渡邊晃：C 言語の通信ライブラリを呼び出す Python ラッパーの提案，平成 30 年度電気関係学会東海支部連合大会論文集，Vol. 2018，講演番号 H2-6，Sep.2018.
（11）渡邉憲士，清水一輝，鈴木秀和，内藤克浩，渡邊晃:iOS の VPN サービスを利用した NTMobile 実装方式の提案，第 81 回情報処理学会全国大会講演論文集，Vol. 2019，講演番号 5V-01，Mar.2019.