アスペクト指向プログラミングを用いた特化による プロセス間通信の高速化の提案

田嶋 孝好[1]　新城 靖[1]

概要：分散システムにおけるプロセス間通信の性能を向上させる手法として、プログラム変換器を用いた特化が知られている。しかし、対象とするプログラムによってはプログラム変換器が利用できない場合がある。そこで、プログラム変換器の代替手法としてアスペクト指向プログラミング (*Aspect Oriented Programming, AOP*) を利用することを提案する。本研究ではプログラム中のプロセス間通信を行う箇所に着目し特化を行う。Java で記述されたプログラムを対象とするため、AOP を行う言語としては Java への AOP 拡張である AspectJ を使用する。HTTPS を用いたプロセス間通信を行う簡単なプログラムを対象に、安全な通信路が利用可能であることを利用して本手法を用いて特化を行った。その結果、同一ホストで実行されることを想定した場合、HTTPS を用いたプロセス間通信を高速化することができた。

1. はじめに

分散システムにおいてプロセス間通信は必要不可欠な要素であり、その性能の向上はとても重要である。プロセス間通信を多用する手法の例として、Microservices[1] が挙げられる。Microservices アーキテクチャは、独立した小さな個々のコンポーネントを組み合わせることで大きな 1 つの分散システムを構築する手法であり、クラウドコンピューティングにおいて、近年注目を集めている。Microservices では、それぞれのコンポーネントが HTTP 上の REST (*REpresentational State Transfer*) でやり取りをしていることが多い。通信を行う際にはセキュリティに注意する必要があり、SSL/TLS が提供する認証や暗号化を利用することでセキュリティの高い通信を行うことができる。

HTTP 通信を行う分散システムでは HTTPS を用いることで通信の盗聴や改ざんを防ぐことが可能になる。ただし暗号化処理には時間がかかり、その結果として分散システム全体の性能を下げてしまうことがある。しかし、分散システムが安全な LAN や VPN で通信を行う場合、暗号化は不要である。よって、実行環境によっては暗号化を行わないことでプロセス間通信を高速化できると考えられる。例えば Microservices では、通信しているコンポーネントが同一ホストで動作している事がある。このような環境では不要な暗号化を行わないことで、プロセス間通信を高速化し Microservices の性能を向上できると思われる。

プロセス間通信に用いられる技術の一つとして RPC (*Remote Procedure Call*) がある。RPC ではデータをクライアントとサーバ間でやり取りするために、メモリ上のデータをネットワークで転送できる形式に変換する。この処理をマーシャリング (*marshaling*) と呼ぶ。本研究では RPC をバイナリ RPC とテキスト RPC に分類して考える。前者はバイナリ形式のデータをマーシャリングする RPC で、後者はテキスト形式のデータをマーシャリングする RPC である。テキスト RPC はバイナリ RPC よりも低速で、通信するデータの量も大きい。Microservices で多用される REST は、テキスト RPC の 1 つとして考えることができる。そのため、マーシャリングのオーバーヘッドが大きいという問題がある。

このような問題を解決する手法として、プログラムの特化 (*specialization*) が知られている。従来、特化を行うにはプログラム変換器がよく利用されてきた。プログラムの利用者は、開発者が作成した汎用プログラムおよび自身が作成したヒントをプログラム変換器に与える。利用者は元のプログラムを直接書き換えずに、特化されたプログラムを得る。従って、元のプログラムに変更が加えられたときに保守が容易になる。

しかし、プログラム変換器が利用できないことや、大きなプログラムでは動作しないことがある。例えば、プログラム変換器の 1 つである部分評価器 Tempo[2] は、C 言語および Java 言語用の部分評価器であるが、大きなプログラムには対応していない。また活発なコンパイラの開発に

[1] 筑波大学
University of Tsukuba

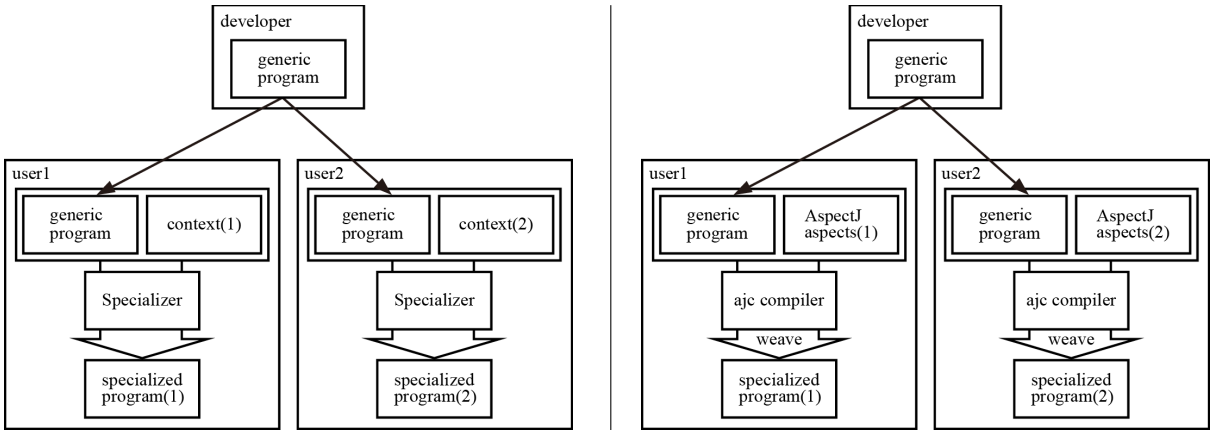

図 1 部分評価器による特化と AspectJ による特化

ついていくことができず、近年保守が行われていない。プログラム変換器を用いずに手動により特化を行うこともできるが、手動による特化は難しく、プログラムの利用者による保守が困難になる。

そこで、本研究では部分評価器の代替手法としてアスペクト指向プログラミング (*Aspect Oriented Programming, AOP*) を利用し特化を行うことを提案する。本研究ではAOP を利用し、プログラムの全体ではなく特定の部分、すなわちプロセス間通信を行う箇所に着目し特化を行う。これによりプロセス間通信を高速化し、分散システムの性能向上を図る。

2. AOP による特化

この章では、AOP による特化の概要について述べる。図 1 の左図は、従来の手法、すなわち部分評価器によりプログラムが特化される様子を表している。最初に利用者である user1 および user2 は開発者 developer が作成した汎用プログラムを手に入れる。user1 および user2 はそれぞれ、汎用プログラムとともに実行環境 context をヒントとして部分評価器に与える。部分評価器としては、代表的なものに Tempo が挙げられる。部分評価器は、context から汎用プログラムの中で静的に計算できるものを事前に計算し、その結果として特化されたプログラムを出力する。context は user1 および user2 で異なるため、同じ汎用プログラムからそれぞれの実行環境に特化された別々のプログラムを得る。

この手法の利点として、プログラムの保守が容易になることが挙げられる。部分評価器を用いた特化はソースコードを直接変更しないため、元のプログラムが変更されたとしても、それに簡単に追随できる。たとえ元のプログラムに大きな変更が加えられて特化ができなくなったとしても、高速なプログラムが得られないだけであり、元のプログラムは動作する。しかし様々な問題により部分評価器が利用できないことがある。1 章で述べたように、部分評価器がメンテナンスされていないことや大きなプログラムでは動作しないという問題がある。

そこで、本研究では部分評価器ではなく AOP を用いて特化を行う。本研究で対象とするプログラムは Java で記述されているため、特化を行うために AOP の Java 実装である AspectJ[3] を使用する。AOP はオブジェクト指向のクラスとは異なる視点によるモジュール化を可能にする。この視点でのモジュールをアスペクト (*aspect*) と言い、アスペクトをソースコードに含めてコンパイルすることを織り込む (*weave*) と言う。AspectJ では、ポイントカットとアドバイスを用いた機構を採用している。ポイントカットはイベントのようなもので、例えばメソッドの実行、クラスのフィールドの参照といったものが、あらかじめ AspectJ の仕様で定められている。ポイントカットを用いて、特定のプログラムが実行されるタイミングを定義することができる。これをジョインポイントと言い、ジョインポイントで行いたい処理をまとめたものをアドバイスと言う。アドバイスの実行タイミングは、ジョインポイントの前である `before()` や、ジョインポイントの後である `after()` などがある。また `around()` を用いることで、ジョインポイントの処理をアドバイスに置き換えることができる。本研究では、これらの機能を用いることでソースコードを変更せずにプログラムを特化する。

AspectJ によりプログラムが特化される様子を表したのが図 1 の右図である。AspectJ を用いた特化も部分評価器を用いた特化と同様に、最初に利用者である user1 および user2 は開発者 developer が作成した汎用プログラムを手に入れる。user1 および user2 はそれぞれ、実行環境に合わせたアスペクトを作成し、アスペクトを汎用プログラムに織り込む。アスペクトは user1 および user2 で異なるため、同じ汎用プログラムからそれぞれの実行環境に特化された別々のプログラムを得る。

プログラムの特化に AspectJ を用いる方法には、部分評価器を用いる従来の手法と同様に、様々な利点がある。開

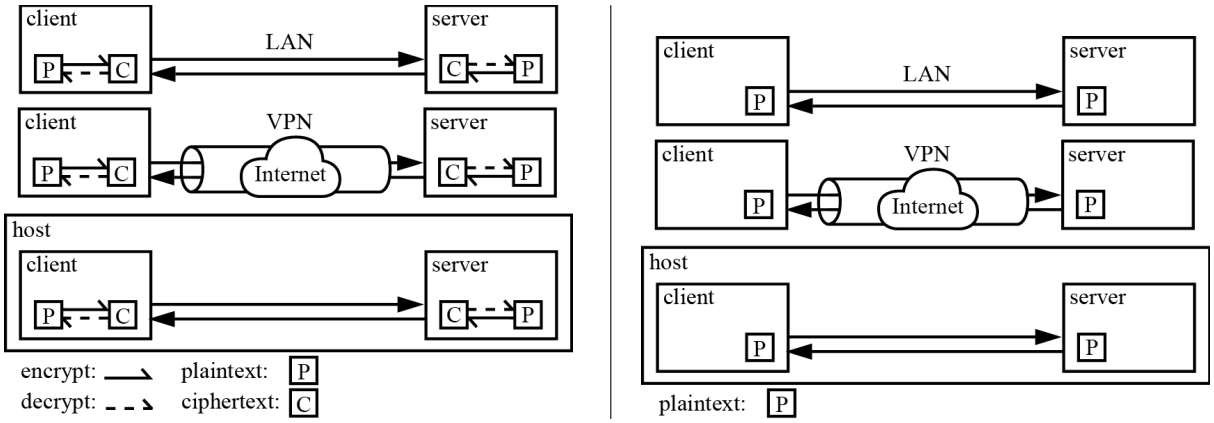

図 2 特化前の HTTPS 通信と特化後の HTTPS 通信

発者は多種多様な実行環境を考慮せずに汎用機能の作成に集中できる。また利用者は実行環境に詳しいため、的確に特化のためのヒントを与えることができる。そしてプログラムを直接変更しないため、利用者はプログラムの保守を容易に行うことができる。部分評価器を用いる従来の手法と比較すると、AspectJ は最新の Java コンパイラに対応しており、部分評価器が動作しないような大きなプログラムでも動作するという利点がある。

本研究では分散システムの通信部分のプログラムを実行環境に応じて特化する。具体的には次のようなことを行う。

- 安全な LAN 内や VPN では暗号化通信を暗号化を行わない通信へ変換する
- 同一ホストで動作している場合は共有メモリを用いたプロセス間通信へ変換する
- REST によるテキスト RPC をバイナリ RPC への変換する

3. HTTPS による通信の特化

分散システムではデータの改ざんや盗聴が行われないように、暗号化通信を用いることが一般的である。図 2 の左図は LAN および VPN における HTTPS 通信の様子を表している。しかし暗号化処理には時間がかかるため、暗号化通信により性能が大きく低下する場合がある。図 2 の右図が示すように、分散システムが安全な LAN 内で通信を行っている場合は、暗号化を行わないことでプロセス間通信の高速化が期待できる。また VPN を利用している場合は、全体で 2 重の暗号化を避けることで高速化が期待できる。

3.1 対象とするプログラム

本研究ではまず、簡単なサーバおよびクライアントを作成し、提案方式で特化が可能であることを確認する。作成したクライアントは HTTPS によりサーバから JSON を受け取る。サーバは Java の HTTP サーバ [4] を用いて作成した。クライアントは `HttpsURLConnection()` を用いて作成した。

図 3 に、作成したサーバのプログラムの主要部分を示す。1 行目および 2 行目でポート番号 `PORT` で接続待ちをするサーバのインスタンスを作成する。4 行目から 13 行目が SSL/TLS に関する設定を行うプログラムである。5 行目から 12 行目で、ファイル名が `KEYSTORE`、パスワードが `PASSWORD` であるキーストアをロードする。キーストアは HTTPS 通信に必要な鍵と証明書を管理するためのファイルで、Java ではこれを使い HTTPS 通信を行う。13 行目でキーストアを管理するキーマネージャを使い SSL/TLS の設定を初期化する。15 行目から 26 行目は HTTPS 通信に必要なパラメータを設定するプログラムである。15 行目および 16 行目で、サーバで有効にする暗号スイートを定義する。このプログラムでは、共通鍵暗号のアルゴリズムが AES であるものを定義している。サーバはクライアントと HTTPS 通信を確立する際に、19 行目から 25 行目のメソッド `configure()` を呼ぶ。このときに 23 行目のメソッド `setCipherSuites()` が呼ばれることで、15 行目および 16 行目で定義した暗号スイートが有効になる。28 行目から 42 行目がパス `/geojson` に対するリクエストハンドラである。`result` はクライアントに送信するデータである。34 行目から 37 行目で HTTP ヘッダを作成する。38 行目から 40 行目で、クライアントへ HTTP レスポンスのボディを送信するための出力ストリームを確保し、HTTP レスポンスのボディを書き込む。44 行目および 45 行目でサーバを別スレッドで起動する。

次に、作成したクライアントのプログラムを図 4 に示す。1 行目から 5 行目で HTTPS 通信に必要なパラメータを設定する。1 行目および 2 行目で接続先の URL のオブジェクトを作成し、このオブジェクトからサーバに接続するためのオブジェクトを作成する。5 行目でリクエストとして `GET` を設定する。サーバプログラムとは異なり、クライアントプログラムでは Java のシステムプロパティを用いて HTTP 通信に使用する暗号スイートを設定する。従って、このプログラムには図 3 の 15 行目、16 行目および 23 行目

```
1  HttpsServer server = HttpsServer.create(
2      new InetSocketAddress(PORT), 0);
3
4  SSLContext ctx = SSLContext.getInstance("TLS");
5  KeyStore ks = KeyStore.getInstance(
6      KeyStore.getDefaultType());
7  ks.load(new FileInputStream(KEYSTORE),
8      PASSWORD.toCharArray());
9  KeyManagerFactory kmf =
10     KeyManagerFactory.getInstance(
11         KeyManagerFactory.getDefaultAlgorithm());
12 kmf.init(ks, PASSWORD.toCharArray());
13 ctx.init(kmf.getKeyManagers(), null, null);
14
15 String[] cipherSuites =
16     { "TLS_ECDHE_RSA_WITH_AES_CBC_SHA256" };
17 server.setHttpsConfigurator(
18     new HttpsConfigurator(ctx) {
19     public void configure(HttpsParameters params) {
20         SSLContext sslctx = getSSLContext();
21         SSLParameters sslparams =
22             sslctx.getDefaultSSLParameters();
23         sslparams.setCipherSuites(cipherSuites);
24         params.setSSLParameters(sslparams);
25     }
26 });
27
28 server.createContext("/geojson",
29     new HttpHandler() {
30     @Override
31     public void handle(HttpExchange t)
32     throws IOException {
33         byte[] result = getResult();
34         t.getResponseHeaders().add(
35             "Content-Type", "application/json");
36         t.sendResponseHeaders(200,
37             result.length);
38         OutputStream os = t.getResponseBody();
39         os.write(result);
40         os.close();
41     }
42 });
43
44 server.setExecutor(null);
45 server.start();
```

図 3 特化対象のサーバプログラム

のような暗号スイートを設定するプログラムはない。7 行目から 19 行目が、サーバと HTTPS で通信を行いサーバから送られた JSON を受け取るプログラムである。8 行目および 9 行目で HTTP レスポンスコードを確認する。10 行目から 18 行目でサーバから送信された HTTP レスポンスのボディのデータを受け取る。HTTP レスポンスのボディの受信は 1 度で終わるとは限らないので、受け取ったレスポンスは一旦 `StringBuilder` オブジェクトに格納し、最後にまとめて文字列にする。21 行目から 23 行目がデータを Java オブジェクトにマッピングするプログラムである。

```
1  URL url = new URL(
2      "https://" + HOST + ":" + PORT + path);
3  HttpsURLConnection conn =
4      (HttpsURLConnection)url.openConnection();
5  conn.setRequestMethod("GET");
6
7  String data = "";
8  if (conn.getResponseCode()
9      == HttpsURLConnection.HTTP_OK) {
10     Reader in = new InputStreamReader(
11         conn.getInputStream());
12     StringBuilder sb =
13         new StringBuilder();
14     while ((count = in.read(buf)) > 0) {
15         sb.append(new String(buf, 0, count));
16     }
17     in.close();
18     data = sb.toString();
19 }
20
21 ObjectMapper mapper = new ObjectMapper();
22 GeoData geodata =
23     mapper.readValue(data, GeoData.class);
```

図 4 特化対象のクライアントプログラム

3.2 NULL 暗号を利用するように特化するアスペクト

本研究ではまず、HTTPS のプロセス間通信を高速化する。そのために暗号化を行わないようにする。そこで、使用する暗号化アルゴリズムを AspectJ を用いて NULL 暗号に変更する。NULL 暗号は SSL/TLS で暗号化を行わないアルゴリズムである。本研究では NULL 暗号を用いる暗号スイートを NULL 暗号スイートと呼び、NULL 暗号スイートとして `TLS_ECDHE_RSA_WITH_NULL_SHA` を用いる。

クラス `HttpsServer` を使った HTTPS サーバは、`setCipherSuites()` を使い有効にする暗号スイートの設定を行う。このメソッドのパラメータを変更することで、NULL 暗号スイートを用いた HTTPS サーバへ特化できる。図 5 に、サーバで有効にする暗号化アルゴリズムを NULL 暗号へ置き換えるアスペクトを示す。図 3 の 23 行目のプログラムが実行されると、このアスペクトに記述したアドバイスが実行される。1 行目と 2 行目で、NULL 暗号スイートを String 型の配列として定義する。6 行目の `proceed()` は特化前のメソッドを実行するメソッドである。`proceed()` の引数を変えると特化前のメソッドの引数も変わる。そこで、引数を NULL 暗号スイート `nullCipherSuites` に変え暗号化通信を行わないプログラムへ変換する。このアスペクトを図 3 に織り込むと、図 3 の 23 行目のプログラムが図 6 の 7 行目のように書き換えられる。

`HttpsURLConnection` を使った HTTPS クライアントは、システムプロパティ `https.cipherSuites` から使用する暗号スイートを取得している。本研究では、`https.cipherSuites` に NULL 暗号スイートを設定する

```
1  String[] nullCipherSuites =
2      { "TLS_ECDHE_RSA_WITH_NULL_SHA" };
3
4  void around() :
5  call(void SSLParameters.setCipherSuites(String[])) {
6      proceed(nullCipherSuites);
7  }
```

図 5 サーバが有効にする暗号スイートを NULL 暗号スイートに特化するアスペクト

```
1  server.setHttpsConfigurator(
2      new HttpsConfigurator(ctx) {
3      public void configure(HttpsParameters params) {
4          SSLContext ctx = getSSLContext();
5          SSLParameters sslparams =
6              ctx.getDefaultSSLParameters();
7          sslparams.setCipherSuites(nullCipherSuites);
8          params.setSSLParameters(sslparams);
9      }
10 });
```

図 6 特化後のサーバプログラム (一部)

```
1  void before() :
2  execution(void Main.main(String[])) {
3      System.setProperty(
4          "https.cipherSuites", nullCipherSuites);
5  }
```

図 7 クライアントが使用する暗号スイートを NULL 暗号スイートに特化するアスペクト

ことで、NULL 暗号スイートを使用するクライアントへ特化する。クライアントが使用する暗号化スイートを NULL 暗号スイートに置き換えるアスペクトを図 7 に示す。クライアントがメインメソッドを実行する直前に、このアスペクトに記述したアドバイスが実行される。3 行目と 4 行目で、`https.cipherSuites` に NULL 暗号スイートを設定する。

3.3 HTTP を利用するように特化するアスペクト

3.2 節では、HTTPS ではあるが、暗号化を行わない方式である NULL 暗号を利用する例を示した。この節では HTTPS の通信を HTTP に置き換える。HTTP では NULL 暗号とは異なり、TLS のハンドシェイクがなくなる。

HTTP サーバはクラス `HttpServer` を使い作成できる。本研究では、`HttpsServer` を `HttpServer` に置き換えることで、HTTPS サーバを HTTP サーバに特化する。図 8 に HTTPS サーバを HTTP サーバに特化するアスペクトを示す。図 3 の 1 行目と 2 行目が実行されるときに、このアスペクトに記述したアドバイスが実行される。2 行目および 3 行目のように `around()` へ引数を与えることで、6 行目で `args()` を使い呼び出し元の引数にアクセスすることができる。この引数を使い、8 行目および 9 行目で HTTP サーバのインスタンスを作成する。15 行目から 18 行目は、図 3 の 16 行目から 25 行目のメソッド呼び出しを行わせないようにするプログラムである。20 行目から 29 行目で HTTP サーバの引数 `path` で指定されたパスに対するリクエストハンドラを登録する。31 行目から 40 行目で HTTP サーバを別スレッドで起動する。

```
1  HttpsServer around(
2      InetSocketAddress addr,
3      int backlog) :
4  call(HttpsServer HttpsServer.create(
5      InetSocketAddress, int))
6  && args(addr, backlog) {
7      try {
8          server =
9              HttpServer.create(addr, backlog);
10     } catch (Exception e) {...}
11
12     return proceed(addr, backlog);
13 }
14
15 void around() :
16 call(void HttpsServer.setHttpsConfigurator(*)) {
17     ; // do nothing
18 }
19
20 HttpContext around(
21     String path,
22     HttpHandler handler) :
23 call(HttpContext HttpServer.createContext(
24     String, HttpHandler))
25 && args(path, handler) {
26     HttpContext ctx =
27         server.createContext(path, handler);
28     return ctx;
29 }
30
31 void around(Executor executor) :
32 call(void HttpServer.setExecutor(Executor))
33 && args(executor) {
34     server.setExecutor(executor);
35 }
36
37 void around() :
38 call(void HttpServer.start()) {
39     server.start();
40 }
```

図 8 HTTPS サーバを HTTP サーバに特化するアスペクト

HTTP クライアントはクラス `HttpURLConnection` を使い作成できる。本研究では、`HttpsURLConnection` を `HttpURLConnection` に置き換えることで、HTTPS クライアントから HTTP クライアントに特化する。図 9 は HTTPS クライアントを HTTP クライアントに特化するアスペクトである。`HttpsURLConnection` インスタンスを作成するときに、このアスペクトに記述したアドバイスが実

```
1  URL around(String spec) :
2  call(URL.new(String))
3  && args(spec) {
4      String httpSpec =
5          spec.replaceFirst("https", "http");
6      try {
7          url = new URL(httpSpec);
8      } catch (Exception e) {...}
9
10     return proceed(spec);
11 }
12
13 URLConnection around() :
14 call(URLConnection URL.openConnection()) {
15     try {
16         conn = (HttpURLConnection)
17             url.openConnection();
18     } catch (Exception e) {...}
19
20     return proceed();
21 }
22
23 void around(String method) :
24 call(void
25     URLConnection.setRequestMethod(String))
26 && args(method) {
27     try {
28         conn.setRequestMethod(method);
29     } catch (Exception e) {...}
30
31     proceed(method);
32 }
33
34 int around() :
35 call(int
36     HttpURLConnection.getResponseCode()) {
37     int code = 0;
38     try {
39         code = conn.getResponseCode();
40     } catch (Exception e) {...}
41
42     return code;
43 }
44
45 InputStream around() :
46 call(InputStream
47     URLConnection.getInputStream()) {
48     InputStream in = null;
49     try {
50         in = conn.getInputStream();
51     } catch (Exception e) {...}
52
53     return in;
54 }
```

図 9 HTTPS クライアントを HTTP クライアントに特化するアスペクト

行される。1 行目から 11 行目で HTTP の URL を作成する。4 行目および 5 行目で URL の https を http に変換する。13 行目から 21 行目で、`HttpsURLConnection` の代わりに `HttpURLConnection` インスタンスを作成する。23 行目から 32 行目でリクエストメソッドを元のプログラムのリクエストメソッドと同一にする。34 行目から 43 行目で HTTP レスポンスコードを取得する。45 行目から 54 行目でレスポンスボディを受け取るための入力ストリームを取得する。

4. 共有メモリを用いた特化

Microservices を構成するコンテナが同一ホストに割り当てられた場合、共有メモリを用いてプロセス間通信を行うことで性能の改善が期待できる。またバイナリ RPC への特化と同様に、テキスト形式ではなくバイナリ形式でマーシャリングを行うことができ、データ量が削減されると思われる。

特化を行うプログラムとして、同一ホストで動作するプログラムを対象にする。本研究ではまず、3 章で述べたプログラムを対象に特化を行う。

4.1 共有メモリを利用するように特化するアスペクト

本研究では、データ量の小さい HTTP リクエストと HTTP レスポンスのヘッダはそのまま HTTPS でやり取りし、データ量の大きい HTTP レスポンスのボディは共有メモリでやり取りするように特化を行う。共有メモリを実装するために `mmap()` システムコールを利用する。本研究では Java からシステムコールを呼び出すために JNA (*Java Native Access*)[8] を用いた。また、共有メモリに書き込むデータは MessagePack[9][10] を使ってマーシャリングを行った。

クライアントに送るデータはサーバのリクエストハンドラが処理する。本研究では、リクエストハンドラが共有メモリに本来のレスポンスのデータを書き込むようにし、書き込んだデータのサイズをクライアントに返すように変更する。図 10 は元のプログラムのリクエストハンドラを、共有メモリを利用するように特化するアスペクトである。3 行目から 7 行目で、JNA を利用して作成した共有メモリを扱うためのクラス `SharedMemory` のインスタンスを作成する。このアドバイスは、HTTPS サーバにリクエストハンドラを登録した直後に実行される。共有メモリの初期化は時間がかかるため、共有メモリの初期化を 1 度だけ行い、サーバが起動している間は同じ共有メモリを利用する。

9 行目から 26 行目が、共有メモリにデータを書き込み、書き込んだデータのサイズをクライアントに返すアスペクトである。図 3 の 31 行目から 41 行目のプログラムが実行されるときに、このアドバイスが実行される。15 行目で、Java オブジェクト `data` を MessagePack を使ってバイナリ形式にマーシャリングし、共有メモリに書き込む。16 行目から 24 行目で、共有メモリに書き込んだデータのサイズをクライアントに送る。クライアントはこのサイズだけ

```
1  SharedMemory shmem;
2
3  after() :
4  call(void HttpsServer.createContext(..)) {
5      shmem = new SharedMemory(
6          SHMEM_PATH, SHMEM_SIZE);
7  }
8
9  void around(HttpExchange t) :
10 execution(void
11     HttpHandler.handle(HttpExchange))
12 && args(t) {
13     try {
14         byte[] data = getData();
15         int size = shmem.write(data);
16         t.getResponseHeaders().add(
17             "Content-Type", "text/plane");
18         byte[] bytesOfSize =
19             String.valueOf(size).getBytes();
20         t.sendResponseHeaders(
21             200, bytesOfSize.length);
22         OutputStream os = t.getResponseBody();
23         os.write(bytesOfSize);
24         os.close();
25     } catch (Exception e) {...}
26 }
```

図 10 共有メモリにデータを書き込みアスペクト

共有メモリからデータを読み出すことになる。

HTTPS クライアントは、サーバから JSON を受け取り Java オブジェクトにアンマーシャリングする。本研究では、サーバから共有メモリに書き込んだデータのサイズを受け取り、共有メモリのデータを Java オブジェクトにアンマーシャリングするように変更する。図 11 に、共有メモリからデータを読み出しアンマーシャリングを行うアスペクトを示す。4 行目から 8 行目で共有メモリをマップする。このアドバイスは HTTPS クライアントインスタンスの作成直後に実行される。クライアントプログラムが終了するまでこの共有メモリが使われる。

10 行目から 17 行目が、共有メモリからデータを読み出すアドバイスである。図 4 の 18 行目のプログラムの実行直後に、このアドバイスが実行される。13 行目で、サーバのレスポンスから共有メモリに書き込まれたサイズを取得する。`strValOfSize` は図 4 の 18 行目に記述された `data` を指す。元のプログラムでは `data` の中身は GeoJSON であるが、図 10 のアスペクトがサーバに織り込まれることで、`data` の中身が共有メモリに書き込まれたデータのサイズに置き換わる。14 行目で共有メモリに書き込まれたデータを読み出す。

19 行目から 31 行目が、共有メモリから読み出したデータをアンマーシャリングするアドバイスである。図 4 の 22 行目および 23 行目のプログラムが実行されたときに、このアドバイスが実行される。図 11 の 22 行目から 30 行目

```
1  SharedMemory shmem;
2  byte[] data;
3
4  after() :
5  call(HttpsClient.new()) {
6      shmem = new SharedMemory(
7          SHMEM_PATH, SHMEM_SIZE);
8  }
9
10 String around() :
11 call(String StringBuilder.toString()) {
12     String strValOfSize = proceed();
13     int size = Integer.parseInt(strValOfSize);
14     data = shmem.read(size);
15
16     return strValOfSize;
17 }
18
19 GeoData around(String strValOfSize) :
20 call(GeoData ObjectMapper.readValue(String, ..)) {
21     GeoData geodata = null;
22     try {
23         ObjectMapper mapper =
24             new ObjectMapper(
25                 new MessagePackFactory());
26         geodata =
27             mapper.readValue(data, GeoData.class);
28     } catch (Exception e) {...}
29
30     return geodata;
31 }
```

図 11 共有メモリからデータを読み込むアスペクト

で読み出したデータのアンマーシャリングを行い、得られた Java オブジェクトを返す。

5. Elasticsearch におけるテキスト RPC の特化

1 章でも述べたように、Microservices では個々のコンポーネントは REST すなわちテキスト RPC で通信を行う。テキスト RPC はバイナリ RPC に比べて低速である。Elasticsearch[7] は Microservices でよく利用され、テキスト RPC で通信を行う。通信部分をバイナリ RPC に置き換えた場合、マーシャリングが高速化され通信速度が向上すると考えられる。また個々のコンポーネントが同じノード上にある場合、RPC ではなく共有メモリを用いることで高速化が期待できる。

5.1 対象とする Elasticsearch とクライアント

本研究では Java で記述されたクライアントと Elasticsearch とのプロセス間通信を高速化する。Elasticsearch のクライアントが Java で記述されている場合、AspectJ を用いることでバイナリ RPC の一つである Java RMI (*Remote Method Invocation*) を利用するプログラムに変更すること

ができる。

図 12 に、クライアントが Elasticsearch から検索結果を受け取るまでの処理の流れを示す。この図では例として、クライアントがパス/apache/log/_search に GET リクエストを送った場合の処理の流れを示している。図 12 において、枠で囲まれているのは Java クラスやスタブである。Java クラスを始点とする破線が時間軸であり、時間軸の進行方向は下である。実線の矢印は、始点にある Java オブジェクトが終点の Java クラスのメソッドを呼び出している様子を表している。実線の矢印の上にはメソッド名が記述されている。メソッドの引数は全て省略している。破線の矢印は、メソッド呼び出しから戻ってくる様子を表している。破線の矢印の上には返り値の実態が記述されている。時間軸にある長方形はメソッドが実行されている時間を表している。

クライアント RestClient がメソッド get() を呼び出すと、Elasticsearch に HTTP リクエストが送られる。Elasticsearch に HTTP リクエストが届くと、Elasticsearch のクラス RestController のメソッド dispatchRequest() が呼び出される。dispatchRequest() が呼び出されるまでの Elasticsearch の処理は省略する。dispatchRequest() は引数に Elasticsearch のクラス RestRequest と RestChannel を取る。RestRequest および RestChannel は、クライアントのリクエストから生成される。RestRequest はクライアントのリクエストを表す。RestChannel はバイト形式の出力チャネルであり、クライアントへレスポンスを送信するために利用される。dispatchRequest() はこれら 2 つの Java クラスを引数として、メソッド executeHandler() を実行する。executeHandler() は引数の RestRequest からハンドラを取得し、そのハンドラのメソッド handleRequest() を実行する。データを検索するリクエストの場合はハンドラとして Elasticsearch のクラス RestSearchAction が得られるため、RestSearchAction の handleRequest() が実行される。検索が完了すると RestChannel は sendResponse() を呼び出し、検索結果を RestResponse のオブジェクトとしてクライアントに送信する。client stub は Elasticsearch から受け取ったレスポンスを Response のオブジェクトにアンマーシャリングし、RestClient へ返す。RestClient はメソッド readEntity() を用いて Response のオブジェクトから検索結果の JSON を取り出す。

5.2 Java RMI を利用するように特化するアスペクト

本研究では、Elasticsearch とクライアントが Java RMI で通信を行うように特化する。そのために、Elasticsearch が起動するときに Java RMI サーバを起動するようにする。そして、REST クライアントがリクエストを送るメソッドをリモートメソッドに置き換える。

リモートメソッドは、クライアントから受け取ったリクエストを Elasticsearch に送り、検索結果をクライアントに返す。しかし、Elasticsearch の検索は非同期で行われるため、リクエストを送った後に検索の完了を待つ必要がある。そこで、本研究ではクラス BlockingQueue を用いて検索の完了を待つ。Elasticsearch が検索結果を書き込むオブジェクトを、RestChannel から BlockingQueue に置き換え BlockingQueue を利用する。図 13 に、BlockingQueue に検索結果を入れるプログラムおよびリモートメソッドを示す。4 行目から 10 行目で、検索結果を BlockingQueue のオブジェクト resQueue に入れる。12 行目から 31 行目がリモートメソッドである。本研究では空のリモートメソッドを作成し、リモートメソッドの処理はアドバイスで記述した。これは、Elasticsearch の RestController を利用して、クライアントから受け取ったリクエストを Elasticsearch に送るためである。15 行目から 18 行目で、クライアントのリクエストから Elasticsearch に送るリクエストとチャネルを作成する。20 行目の restController が RestController のオブジェクトである。dispatchRequest() にリクエストとチャネルを与えて呼び出すことで、Elasticsearch の検索が始まる。23 行目から 26 行目で検索結果を BlockingQueue から取り出す。29 行目で検索結果をディープコピーし、30 行目でディープコピーしたオブジェクト result をクライアントに返す。

6. Android におけるテキスト RPC の特化

Android は Google が開発した携帯端末のためのプラットフォームである。Android 上で動くアプリケーションは Java で記述されている。

Android アプリケーションがクラウド上のサーバと通信する場合にテキスト RPC が使われていることがある。これをバイナリ RPC に変換することで CPU 処理が削減され、バッテリーの消費が抑えられると考えられる。また通信するデータ量が削減されることで通信時間が削減されると考えられる。

特化を行うプログラムとして、Android-サーバ間を REST で通信するプログラムを考えている。このプログラムにアスペクトを織り込むことで、REST からバイナリ RPC を用いたプログラムに特化する。Android では Java RMI を使用することができないため、gRPC[5] というバイナリ RPC を用いる。gRPC でやり取りするデータ構造の定義やインタフェースの定義には Protocol Buffers[6] を用いる。

7. 実験

3 章、4 章、そして 5 章で述べた特化について、特化前と特化後の通信時間を測定した。

最初に、次の 4 つのプログラムについて通信時間を計測した。

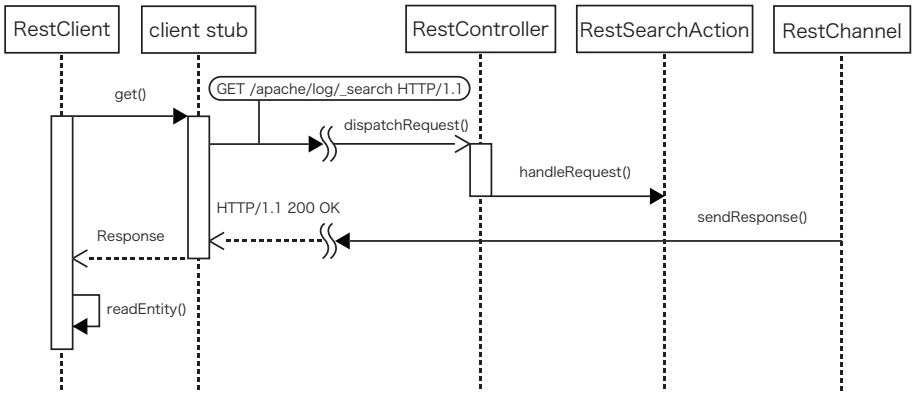

図 12 クライアントが Elasticsearch から検索結果を受け取るまでの処理の流れ

```
1  BlockingQueue<RestResponse> resQueue =
2      new ArrayBlockingQueue<>();
3
4  void around(RestResponse response) :
5  execution(void RestChannel.sendResponse(RestResponse))
6  && args(response) {
7      try {
8          resQueue.put(response);
9      } catch (Exception e) {...}
10 }
11
12 ESResult around(String url, String query) :
13 onGetRequest() && args(url, query) {
14
15     RMIRequest request
16         = new RMIRequest(url, query);
17     RMIChannel channel
18         = new RMIChannel(request);
19
20     restController.dispatchRequest(
21         request, channel);
22
23     RestResponse response;
24     try {
25         response = resQueue.take();
26     } catch (Exception e) {...}
27
28     /* deep copy of the searching result */
29     ESResult result = getDeepcopy(response);
30     return result;
31 }
```

図 13 Elasticsearch に織り込むアスペクト

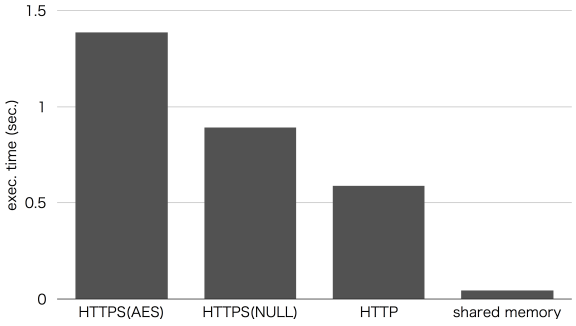

図 14 HTTPS によるプロセス間通信の特化による高速化の結果 (同一ホスト内)

- 元の簡単な HTTPS のプログラム (AES を使用する暗号スイート利用)
- NULL 暗号を利用するように特化したプログラム
- HTTP を利用する (SSL/TLS を使用しない) ように特化したプログラム
- 共有メモリを利用するように特化したプログラム

特化前のプログラムおよび特化後のプログラムは、クライアントとサーバが総務省の地理データ [11] をやり取りする。HTTPS のプログラム、NULL 暗号を利用するように特化したプログラム、そして HTTP を利用するように特化したプログラムは、データを JSON にマーシャリングする。クライアント-サーバ間を行き来する JSON の大きさは約 105MB である。共有メモリを利用するように特化したプログラムは、データをバイナリにマーシャリングする。実験環境は、JDK が Java 1.8、CPU は Intel Core i7-3820 3.60GHz を用いた。クライアントとサーバは同一ホストで実行した。実行時間は、特化前および特化後ともに 100 回計測した実行時間の平均を取った。ただし、最初の通信では初期化などの処理を伴うため実行時間には含めなかった。

図 14 に実験の結果を示す。y 軸は通信時間をミリ秒単位で表している。図 14 が示す通り、特化後のプログラムは特化前のプログラムに比べて通信時間が短くなっていることがわかる。これは、NULL 暗号、HTTP、共有メモリを利用するように特化したプログラムでは、暗号化処理

を行わなくなった影響である。さらに、特化後のプログラムは、NULL 暗号を利用するように特化したプログラム、HTTP を利用するようにしたプログラム、共有メモリを利用するように特化したプログラムの順に、通信時間が短くなっている。これは、暗号化処理や SSL/TLS ハンドシェイク、そして共有メモリを利用する場合は TCP のオーバーヘッドが削減されたことによる。また、データをバイナリにマーシャリングすることで、クライアント-サーバ間でやり取りするデータが小さくなったことも挙げられる。総務省の地理データを JSON にマーシャリングするとサイズは約 105MB であった。一方、バイナリにマーシャリングするとサイズは約 65MB であった。

次に、Elasticsearch を Java RMI に特化したときの実行時間を計測した。特化前のプログラムおよび特化後のプログラムは、クライアントがサーバから総務省の地理データの検索結果を受け取る。Elasticsearch にはあらかじめ、総務省の地理データを JSON に変換したものを 4316 件登録した。特化前のプログラムは検索結果を JSON でやり取りし、特化後のプログラムは検索結果をバイナリでやり取りする。

実験の結果として、特化後の Elasticsearch の方が特化前の Elasticsearch よりも通信速度が低くなった。やり取りするデータのサイズは特化後の Elasticsearch の方が大幅に少なかった。通信速度が低下した理由として、ディープコピーを行っていることが挙げられる。つまり、サーバがクライアントに送る全てのオブジェクトをコピーしている。これは、Elasticsearch が生成するオブジェクトはプライベートなフィールドを含んでおり、Java RMI の方法では単純にマーシャリングできなかったためである。これにより、マーシャリング可能なオブジェクトにディープコピーをする必要があった。

8. 評価

本研究の目的は、分散システムのプロセス間通信を高速化し、性能を向上させることである。プロセス間通信を高速化するために AOP を用いてプログラムの特化を行った。本研究で対象としたプログラムは Java で記述されているため、AOP を行うために AspectJ を用いた。本研究では、以下のような特化を行いプロセス間通信の高速化を試みた。

- HTTPS による通信を、安全な LAN では暗号化処理を行わない通信に特化する
- 同一ホストで通信を行う際に共有メモリを利用するように特化する
- テキスト RPC をバイナリ RPC に特化する

これらの特化は利用者が行うため、開発者は実行環境を考慮せずに汎用的なプログラムを作成することができる。また開発者が作成したプログラムは直接変更されず、アスペクトは開発者が作成したプログラムと完全に独立している。よって、利用者は容易にプログラムの保守ができる。

HTTPS による通信を暗号化処理を行わない通信に特化する実験では、特化する通信手段によってアスペクトの記述量が変わった。NULL 暗号へ特化する場合は、有効にする暗号スイートを NULL 暗号スイートとするだけでよい。HTTP へ特化する場合は、HTTP への特化に加えて HTTPS 固有の設定を無効にするアスペクトが必要である。共有メモリへ特化する場合は、アスペクトだけでなく共有メモリを利用するプログラムを用意する必要がある。

本研究では、特化するプログラムやアスペクトの種類によって通信速度が改善されない場合があった。Elasticsearch の特化では、特化後の Elasticsearch の方が通信速度が低くなってしまった。しかし、通信するデータのサイズの削減には成功した。そして、Elasticsearch のような大きいプログラムでも AspectJ を用いることで特化を行うことができることを確認した。

9. 関連研究

Tempo は C 言語および Java 言語用の部分評価器である。Tempo を用いた研究として Sun RPC の最適化に関する研究 [2] や POSIX 準拠のマルチスレッドプログラムに関する研究 [12] がある。これらの研究は、実行前に行う静的な部分の計算や、関数呼び出しの際に実行される命令の削減により性能の改善を行っている。

1 章でも述べたように、部分評価器の保守は容易ではない。本研究では、保守が容易なアスペクト指向プログラミングの処理系を利用して特化を行う。

分散システムのパフォーマンス解析を行うフレームワークとして Mystery Machine[13] がある。Mystery Machine は ÜberTrace と呼ばれるトレースシステムが生成するトレースを用いて、因果関係モデルを作成する。その因果関係モデルを基に、分散システムのオーバーヘッドである箇所を探し、開発者が人手で最適化を行う。

本研究では、プログラムの高速化は開発者ではなく利用者が行う。利用者は自身の実行環境を熟知しており、それを利用して汎用プログラムを高速なプログラムに変換して実行する。

オフローディングを用いた Android アプリケーションの性能改善に関する研究 [14] がある。この研究では、最初に Android アプリケーションの中でオフローディング可能なクラスを検知する。そして、元のプログラムをオンデマンドにオフローディングを行うプログラムへ変更する。オフローディング可能なクラスの検知やプログラムの変更には、この研究で開発した DPartner[15] と呼ばれるプログラムを利用している。

本研究はオフローディングは利用せずに、プロセス間通信に着目し性能向上を図る。そしてプログラムを変更する手法として AOP を用いる。

10. おわりに

この論文では、AOP を用いた特化によるプロセス間通信の高速化について述べた。安全な LAN 内や VPN における暗号化通信を、暗号化を行わない通信に変換することで通信速度の向上が期待される。また、テキスト RPC からバイナリ RPC への特化および共有メモリの利用により通信速度を高速化することで、Microservices や Elasticsearch の性能向上が期待される。そして携帯端末では、特化により CPU 処理が削減され、バッテリー消費が抑えられることが期待される。本研究では、簡単な HTTPS サーバおよびクライアントを対象として、提案方式に基づき特化を行った。その結果、同一ホスト内の通信を高速化することを確認した。Elasticsearch を対象として特化を行ったが、オブジェクトのコピーの問題により通信速度は高速化されなかった。

今後の予定として、Android アプリケーションに関する実験を行い性能評価を行う。具体的には、テキスト RPC をバイナリ RPC へ変換することでどのような影響があるのかを調べる。そしてアプリケーションに対して AOP による特化を行うことで、アプリケーションの性能にどのような影響を与えるのかを調査する。

参考文献

[1] S. Newman, Building Microservices - Designing Fine-Grained Systems., O'Reilly Media, 2016.

[2] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, A. Goel, "Fast, Optimized Sun RPC Using Automatic Program Specialization," Proc. 19th IEEE Int'l Conf. Distributed Computing Systems (ICDS '98), pp. 249-258, 1998.

[3] The Eclipse Foundation. "The AspectJ Project," http://www.eclipse.org/aspectj/, accessed: 2017-11-07.

[4] "com.sun.net.httpserver (Java HTTP Server)", https://docs.oracle.com/javase/8/docs/jre/api/net/httpserver/spec/com/sun/net/httpserver/package-summary.html, accessed: 2017-11-07.

[5] Google, "gRPC - a high performance, open-source universal RPC framework," https://grpc.io/, accessed: 2017-11-07.

[6] Google, "Protocol Buffers - Google Developers," https://developers.google.com/protocol-buffers/, accessed: 2017-11-07.

[7] Elastic, Inc. "Elasticsearch: Search & Analyze Data in Real Time," https://www.elastic.co/products/elasticsearch, accessed: 2017-11-07.

[8] "Java Native Access (JNA)", https://github.com/java-native-access/jna, accessed: 2017-11-07.

[9] "MessagePack: It's like JSON. but fast and small.," https://msgpack.org/, accessed: 2017-11-07.

[10] "jackson-dataformat-msgpack," https://github.com/komamitsu/jackson-dataformat-msgpack, accessed: 2017-11-07.

[11] 地図で見る統計（統計 GIS）, http://e-stat.go.jp/SG2/eStatGIS/page/download.html, accessed: 2017-11-07.

[12] Y. Shinjou, C. Pu, "Achieving Efficiency and Portability in Systems Software: A Case Study on POSIX-Compliant Multithreaded Programs," IEEE Transaction on Software Engineering, Vol. 31, No. 9, pp. 785-800, September 2005.

[13] M. Chow, D. Meisner, J. Flinn, D. Peek, T. F. Wenisch, "The Mystery Machine: End-to- end Performance Analysis of Large-scale Internet Services," 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14), pp. 217-231, October 6-8, 2014.

[14] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, S. Yang, "Refactoring Android Java Code for On-Demand Computation Offloading," OOPSLA'12. October 19-26, 2012, Tucson, Arizona, USA.

[15] DPartner, https://code.google.com/archive/p/dpartner/, accessed: 2017-11-07.