Vol.2020-CSEC-89 No.19
Vol.2020-IOT-49 No.19
2020/5/15

Weave tile: 1万監視ターゲットの準リアルタイム可視化

阿部　博[1,a)]　古澤　徹[1,b)]　岡田　和也[2,c)]

概要：コンテナ基盤の流行により，多くのコンテナがマイクロサービスのインスタンスとして利用されている．これにより既存の仮想マシンと比較した場合，運用者が管理しなければならないインスタンスの数が大幅に増加したことを意味する．また，IoT やエッジコンピューティング，コネクティッドカーというキーワードに代表されるように，大量のインスタンスが積極的にネットワークへと接続し，データの送受信を行う世界となってきた．本研究では，運用者が既存監視システムでは監視を行いにくい，比較的規模の大きな 1 万ターゲットを準リアルタイムに監視を行い，監視結果を可視化する仕組みである「Weave tile」を提案する．

キーワード：監視，準リアルタイム処理，可視化，ビッグデータ

Weave tile: Near real time processing and visualization for massive monitoring target

Hiroshi Abe[1,a)]　Toru Furusawa[1,b)]　Kazuya Okada[2,c)]

Abstract: Many containers use as instances of microservices due to the popularity of container infrastructure. The number of instances that operators must manage has increased when compared to managing existing virtual machines. Also, a large number of instances, such as IoT devices, edge computing equipment and connected cars, actively connect to the network, and send and receive data. In this study, operators monitor relatively large 10,000 targets in near real-time, which is difficult for operators to monitor with existing monitoring systems. We propose "Weave tile", a mechanism to visualize monitoring results in near real-time.

Keywords: Monitoring, Near real time processing, visualization, big data

1. 背景と目的

コンテナ基盤の流行により，多くのコンテナがマイクロサービスのインスタンスとして利用されている．これにより既存の仮想マシンと比較した場合，運用者が管理しなければならないインスタンスの数が大幅に増加したことを意味する．また，IoT やエッジコンピューティング，コネクティッドカーというキーワードに代表されるように，大量のインスタンスが積極的にネットワークへと接続し，データの送受信を行う世界となってきた．これによりネットワークへと接続されるインスタンスの監視が今までのように監視システム側からアクティブに行うのではなく，エージェントを監視インスタンス内へと用意し，監視システム側でデータを受け取り監視を行うパッシブ型監視システムやモニタリング Software as a Service (SaaS) が増加してきた．

アクティブな監視やエージェント型監視を用いる場合，監視システム運用者は主にダッシュボードを用いて監視を行いたい情報を集約し，運用に役立つように業務を最適化する．しかしながら，監視ターゲットが大量に存在する場合には，情報を全て一元的に表示することは難しく，統計情報等を用いたダッシュボードの表示を行うことが多い．

[1] トヨタ自動車株式会社
[2] 東京大学
[a)] hiroshi_abe_af@mail.toyota.co.jp
[b)] toru_furusawa@mail.toyota.co.jp
[c)] okada@ecc.u-tokyo.ac.jp

ⓒ 2020 Information Processing Society of Japan

大部分の監視業務は統計情報のみで最適化可能であるが，個々のインスタンスに注力して監視を行う必要がある場合，かつそのインスタンスが大量のエンティティである場合には表示方法を工夫しなければ，監視情報の欠落が発生し障害への対応が遅延する可能性がある．

監視対象のターゲット数が多くなった場合には，ダッシュボードを用いて統計的な情報を基に監視を行う場合がある．特に監視ターゲット数が莫大な場合には，個々の監視対象をダッシュボードで視覚的に見ることはせず，異常時に特定ターゲットが異常であるという情報のみを表示する場合が多い．しかしながら，特定ターゲットの異常が広範囲に発生した場合に異常状態の通知が大量に行われ，特定情報のみを表示するという目的であった運用は破綻する可能性がある．部分的な障害や大規模な障害を直感的に視覚的に捉えるには，監視ターゲット全体を見る行為が必要になる．本研究では「木を見て森を見ず」な状態や「森を見て木を見ず」ではなく，「木を見て森を見る」監視を視覚的に実現する．

次に監視の間隔を考える．監視間隔は監視システムがデータを受信し処理を行う時間を考慮し，監視ターゲット数や監視項目の最大値が計算機スペックによって決定する．低いスペックの計算機では，大量のデータを受信するだけで，CPU リソースやネットワーク帯域が頭打ちとなる．可能である限り，監視ターゲットと監視項目を大量に集約し，少ない計算リソースで最大の集約効率をあげることが利用コストを考える上では有益である．クラウド環境であれば，監視を行う計算機のスペックはスケールイン・スケールアウトする戦略を取りやすいが，スケールイン・スケールアウトのどちらをとっても利用コストの増加は免れない．

本研究では可能な限り，計算機の性能を最大限に引き出し監視項目を増やすアプローチをとる．目標として，1 万ターゲットのステータス監視データを毎秒受信し，並行して全監視ターゲットを視覚情報として数秒遅れの準リアルタイム処理にて視覚化する．このシステムを「Weave tile」と名付ける．

2. 関連研究

オンプレミス環境で利用されることが多い，SNMP を用いた Zabbix[1] や Nagios[2] といった監視システムでは，データ取得のポーリング間隔を 1 分から 5 分のように設定することができる．これはデータ収集ターゲットの数により収集間隔を設定することと，収集するデータの解像度により間隔を設定することができる．これらの監視システムではエージェントから毎秒データを受け取ることや，ポーリングを 1 秒単位で行うことは考えられておらず，分単位という比較的間隔の長い幅でのデータ取得を行う．逆にリアルタイム性を求める場合には SNMP Trap を用いた情報取得を行い，管理者へと通知するアプローチをとる．

次に SaaS サービスである DataDog[3] や Mackerel[4] といった監視システムでは，SNMP を利用したポーリングシステムとは違い基本的に監視対象にインストールされたエージェントベースでの監視が行われる．これにより情報収集間隔はエージェントの起動タイミングで制御される．アプリケーションパフォーマンスモニタリングを行うことも可能な場合が多く，5 秒単位からといった比較的細かい解像度で監視データを送信することが可能となる．

監視データの配送には，SNMP を用いた閉域網での暗号化を行わない UDP を用いた収集や，エージェントによる TCP と TLS を用いた暗号化データの送信などがある．また，データを取得後すぐにデータ送信を繰り返すために，データが送信されることを保証する MQTT を利用する場合や，さらに MQTT を改良し，階層化（デバイス/エッジ/クラウドの 3 層構造）や P2P の技術を応用した分散型 MQTT などの研究が行われている [5]．

収集されるデータは，RRDTool[6] の様に，名前に示されるような循環した領域を決められた範囲で上書きしていくデータ構造が用いられたり，データベースへとスキーマを定義した上で格納される場合や，時系列データベースの様に時系列に特化したデータ保存手法が利用される．時系列データ保存を最適化するために，複数のデータベースを用いた HeteroTSDB[7] という実装も存在する．保存されるデータがログ形式の場合には，Hayabusa[8] の様な SQLite ファイルへの書き込みと並列検索手法が用いられ場合もある．

データの蓄積や処理，取得間隔には複数のメトリックが考えられる．IoMT[9] というキーワードにおいてサービス事例が分類されており，どのサービスがどの程度の反応速度で応答するべきか考察されている．サービスの応答速度の視点でデータの一貫性や処理性能を考慮する必要があり，それによってデータの収集方法や蓄積方法を選択する必要がある．

よりリアルタイムに近いデータ解析や可視化に関して，Open Source Software (OSS) として NETDAT[10] が公開されている．

3. 設計

3.1 アーキテクチャ

図 1 に Weave tile のアーキテクチャを示す．Weave tile では，監視対象となるクライアントからハートビートパケットを受け取る．ハートビートパケットは，クライアントのステータスを記録するパケットであり，1 秒に 1 回クライアントより送出される．ハートビートパケットを受け取る仕組みとして reciever がデーモンプロセスとして動作する．reciever はクライアントから受け取ったハートビートパケットをストレージへと記録する．ハートビートパ

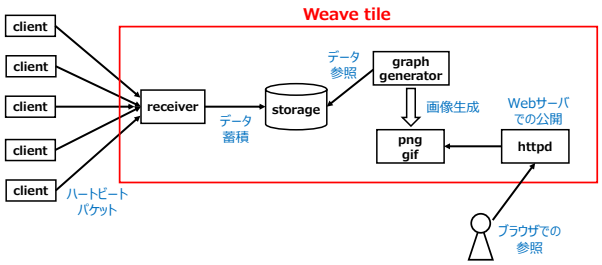

図 1　Weave tile のアーキテクチャ
Fig. 1　Weave tile architecture

ケットは以下の形式で表される．

- 時間（unixtime）
- クライアント識別子（整数値）
- ステータス（0,1,2 の数値）

ここで表されるステータス情報は，以下 3 種類に定める．

- 0: 正常
- 1: 部分故障
- 2: 異常

将来的にさらにステータスを追加することは可能ではあるが，本研究では 3 種類以上の意味は持たせない．

次に，ハートビートパケットには UDP を選択する．TCP を選択しない理由には，接続の担保のため発生するオーバヘッドと暗号化オーバヘッドの 2 点をあげる．TCP を用いる場合には，サーバとの接続性が担保される．しかしながら接続性は維持されるが 3-way handshake のオーバヘッドがかかり，かつネットワークが不安定になった場合には輻輳が発生し，データの到達に意図せず遅延が発生する場合がある．また暗号化オーバヘッドに関しては，本実験を行うネットワークは閉域であると仮定し，本課題からは条件として排除する．UDP を用いることで，TCP と比較した場合にデータ到達性の保証がなくなるが，クライアントは 1 秒に 1 度ハートビートパケットを送信するためデータの損失が不定期に発生した場合に，監視自体に影響はないと判断する．UDP を用いて発生する中長期の期間，ハートビートパケットが到達しなかった場合の監視についても本研究では対象外とする．

graph generator は蓄積されたハートビートパケットを読み込み unixtime で表記された該当時間 1 秒単位の画像を PNG ファイルとして生成する．生成された PNG ファイルは，最終的に結合されアニメーション GIF ファイルとしてまとめあげられ保存される．保存されたアニメーション GIF ファイルは，Web サーバ（httpd サーバ）のコンテンツとしてブラウザに表示可能な形式で提供される．

生成される PNG 画像は，タイル状の画像としてステータス情報を表現する．タイル状の画像として表現する理由

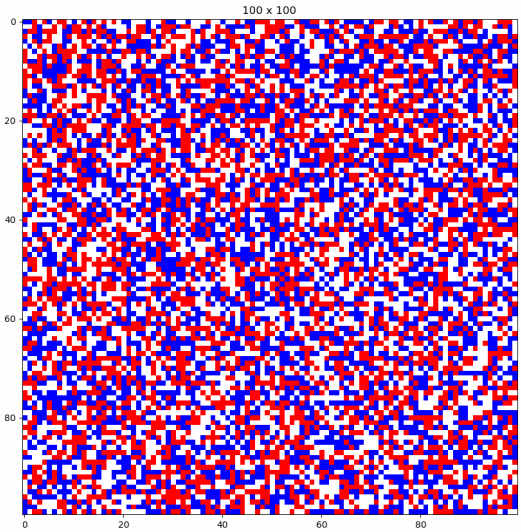

図 2　生成されるタイル画像
Fig. 2　Generated tile image

に関しては，多数の監視対象を同時に視認することが可能であることや，1 万監視ターゲットという目標値に対して 100 x 100 の形状で表現可能な形式として画像を生成しやすい点などがあげられる．図 2 に，生成される画像イメージをあげる．ここでの配色は，ステータスの数値情報により変化する．

ハートビートパケットを監視したいユーザは，Web サーバへとアクセスすることで，クライアントのステータス情報をタイル画像として確認することができる．

4. 実装

本研究では reciver を独自実装せず，すでに最適化されているであろう syslog プロトコル [11] に乗っ取った実装を利用することで，ハートビートパケットを効率的に受信し，ファイルへと高速に書き込む手法として rsyslogd[12] を選択する．

4.1 クライアント

本実装でクライアントは UDP パケットを reciver へと送信する．クライアントは，1 秒間に 1 ハートビートパケットを送信する．本実験では 1 万クライアントからのハートビートパケットの受信を reciver は想定するため，秒間 1 万パケット (10,000pps) の送信をクライアントから行う．このようなパケット送信であれば，規模が大きい場合にはテスターを用いるのが最適ではあるが，本実験は比較的小規模のため，1 クライアントプログラムより 10,000pps のパケットの送信を実装した．

図 3 は，クライアントサンプルプログラムの抜粋であり，

```
func main() {
    cpus := runtime.NumCPU()
    runtime.GOMAXPROCS(cpus)

    conn, err := net.Dial("udp", "reciver ip addr:514")
    if err != nil {
        os.Exit(1)
    }
    defer conn.Close()

    var utime = スタートのunixtime
    var target = 10000
    var sec = 60

    rand.Seed(time.Now().Unix())

    for i :=0; i < sec; i++ {
        for ii :=0; ii < target; ii++ {
            msg := fmt.Sprintf("%d %d %d", utime, ii, rand.Intn(3))
            go conn.Write([]byte(msg))
        }

        utime++
        time.Sleep(time.Millisecond * 送信間隔遅延ミリ秒)
    }
}
```

図 3 クライアントソースコード
Fig. 3 Client code

golang を用いて実装を行ったコードである．10,000pps を実現するため，CPU のコア数の最大値分のプロセスを生成し，並列に UDP パケットを生成し出力を行う．ステータスは，適宜ランダムに生成される．1 秒間隔でパケット送信を行う為に，必要数秒送信を継続する場合には，適宜 1 秒弱の遅延を挿入する．

4.2 reciever

本研究では，receiver として rsyslogd を選択した．UDP パケットを受信し，ファイルへと書き込むという動作であれば，rsyslogd を用いる必要がなく自前で実装可能である．rsyslogd を採用した理由は「コモディティソフトウェア」であることと「最適化されたソフトウェア」である 2 点である．

rsyslogd は Ubuntu や CentOS をはじめ多くの Linux ディストリビューションで標準的な syslog を扱うソフトウェアとしてバンドルされている．また syslog プロトコル [11] は，Internet Engineering Task Force (IETF) により Request for Comments (RFC) として定義され，広く多くのソフトウェアやアプライアンスで利用され，ログを出力する，または収集する上で標準的に利用されるプロトコルである．これらを考慮すると，rsyslogd を利用することは標準化に従ったコモディティソフトウェアを採用することを意味する．

```
$IncludeConfig /etc/rsyslog.d/template.conf
:fromhost-ip, isequal, "client ip address"  -/var/log/remote-udp.log;status
& ~
```

図 4 rsyslogd の設定
Fig. 4 rsyslogd configuration

次に rsyslogd は大量のログを受けるために，多くの仕組みが備わっている．rsyslogd 自身にキューイングの仕組みが組み込まれており，メッセージの取りこぼしが少ない．さらに環境に合わせ，キューの長さや処理のパラメータを変化させることで，性能を大幅に向上させることができる．受信ソフトウェアを自身で作成する場合，キューの実装，トラブル時のキューデータの保存や復元など，本システムとして目指したい部分とは違う部分に労力が割かれるため，本研究での受信プログラムの自作は見送る．将来的に，rsyslogd でパフォーマンスの問題が発生した場合には再考を行う．

rsyslogd 自身のチューニングは行わず，クライアントから受け取ったパケットを syslog パケットとして擬態を行い，ログファイルへと書き込みを行う．rsyslogd の設定は図 4 となる．また，rsyslogd.conf から読み込まれるテンプレートファイルを以下の通り定義する．

```
$template status, "%HOSTNAME% %syslogtag% %msg%\n"
```

この定義により，クライアント IP アドレスから到着するハートビートパケットは syslog パケットと同等に処理され，「unixtime クライアント ID ステータス」の形式でファイルへと記録される．

具体的に，rsyslogd によりログファイルへと記述される内容は以下のテキスト形式となる．

```
1578458878 1  2
1578458878 16  1
1578458878 7  2
1578458878 8  0
...
```

これは，テンプレートファイルの定義に基づいたフォーマットであり，CSV と同等の Space-Separated Values (SSV) 形式となる．クライアントが生成する UDP パケットは並列送信されているため，到着の順序保証はなくログに書き込まれ，クライアント ID は到着順にランダムに記録さ

ⓒ 2020 Information Processing Society of Japan

れる．

4.3 ログのローテーション

receiver (rsyslogd) でログとして CSV データを格納するファイルは，Linux のログローテーションの仕組み（logrotate デーモン）を用いて 1 分単位で新しいファイルへとローテートされる．これは 1 つのログファイルへの追記を行い続けることにより，ログデータ自体が巨大化することを防ぐ意図がある．この仕組みにより，1 分に 1 ファイル新しい CSV データファイルが生成されることになる．

4.4 graph generator

graph generator には処理が複数存在する．以下に，graph generator の具体的な作業フローを明記する．

(1) ログファイルからのデータの読み込み
(2) データのソート
(3) 指定された秒単位のデータの抜き出し
(4) データを配列へ転換
(5) 配列データから PNG 画像を生成
(6) 生成済み PNG 画像群を GIF アニメーションへと変換

本実験では，上記処理を実現するために graph generator を Python 3 で実装する．(1), (2), (3), (4) の処理を行うため，Python の Pandas ライブラリ [13] を用いる．Pandas は，Python でデータ解析を支援する機能を提供するライブラリである．特に，数表および時系列データを操作するためのデータ構造と演算を提供する．

図 5 に，(1) から (4) までの処理を示す．ここでは，Pandas Dataframe の機能を用いファイルから読み出したデータに'time', 'id', 'status' の 3 つのカラム名を付ける．次にデータを時間とクライアント識別子でソートし，グラフ生成関数 (create_graph) へとデータと引数の時間を渡す．グラフ生成関数は，受け取ったパラメータの中から引数として渡された該当時間のデータを抜き出し，100 x 100 の配列へとデータを変換する．

次に，(5), (6) の画像生成処理へと移る．(5) の処理で生成される画像は，(4) の処理で秒単位の時間のパラメータ毎に生成され，PNG ファイルとして保存される．この処理は，Matplotlib の機能を用いて実装を行った．図 5 の例では，range(10) で 10 回のループを処理するため，スタートの unixtime から 10 秒後までの画像が生成される．

(6) に関しては，図 6 に示すように生成された画像をまとめあげるために Python の Pillow ライブラリを用いて GIF アニメーション画像へと変換する．

以上が，graph generator の大きな処理の流れとなる．

4.5 httpd

graph generator によって生成された GIF 画像は，Web サーバ上に公開される．本研究では Web サーバとして

```
1  fname = 対象のログファイルパス''
2  df = pd.read_csv(fname, names=('time', 'id', '
       status'), sep=' ', skipinitialspace=True)
3  df = df.sort_values(['time', 'id'])
4
5  t = スタートのunixtime
6  for i in range(10):
7      create_graph(df, t)
8      t += 1
9
10 files = sorted(glob.glob('./img/*.png'))
11 images = list(map(lambda file: Image.open(file
       ), files))
12
13 images[0].save('./img/out.gif', save_all=True,
       append_images=images[1:], duration=1000,
       loop=0)
14
15 def create_graph(df, time):
16     df2 = df[df.time == time]
17     status = df2.status
18
19     Z = status.values.reshape(100,100)
20
21     ax0.imshow(Z, cmap=cmap)
22     plt.savefig('./img/' + str(time) + '.png')
```

図 5 ログの読出しとデータ加工，画像生成
Fig. 5 Log read, processing, generate images

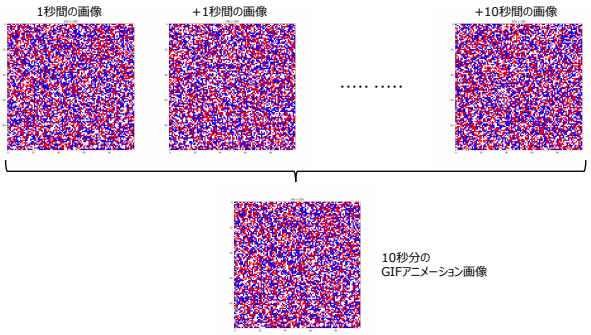

図 6 GIF アニメーション生成
Fig. 6 Generate GIF animation

Apache2 を利用した．Apache2 側で特に動的な処理は行わずに，graph generator により生成された画像を，ブラウザで表示できる形で公開する．

4.6 ブラウザ

graph generator により，準リアルタイムに GIF 画像が生成され置き換えられる．ユーザが利用するブラウザは，新しく生成された画像を自動的に読み込むことはない．そこで，本件研究ではユーザが利用するブラウザにプラグイ

表 1　評価環境
Table 1　Environment

ハードウェア	SuperMicro E300-9D
CPU	Intel Xeon D-2146NT (2.30GHz/8 cores)
Memory size	64 GB
Disk size	NVMe SSD 1TB
OS	Ubuntu 18.04.3 LTS

表 2　送信パケットと受信ログ行数
Table 2　Packet per seconds and received logs

秒数	pps	ログ行数
1 秒	10,000pps	10,000 行
10 秒	100,000pps	100,000 行
20 秒	100,000pps	200,000 行
30 秒	300,000pps	300,000 行
40 秒	100,000pps	400,000 行
50 秒	100,000pps	500,000 行
60 秒	600,000pps	600,000 行

ンを導入し，5 秒に一度 Web サーバへと設置された GIF 画像をリロードする処理を行う．これにより，ユーザは切れ目なく 1 秒間隔で監視が継続しているように画面を確認することができる．

5. 評価

5.1 評価環境

評価環境を表 1 に示す．本環境は CPU に物理 8 コア，HyperThreading が有効な場合に 16 コアの処理が可能な Xeon D-2146NT を搭載する．またメモリは 64GB, ディスクに NVMe で接続された SSD を 1TB 搭載している．クライアント，サーバ共に同じスペックのマシンを利用し，1Gbps のネットワークスイッチに接続をしている．

5.2 パケット受信とログ保存

クライアントから 10,000 pps の UDP パケットを送信し，syslog データとして受信しログファイルへと保存可能かを評価した．1 秒から 60 秒までの UDP パケットをクライアントから送信し，サーバ側で syslog として記述できたログ行数を表 2 で表す．

クライアントから送信した UDP パケットと，サーバ側で rsyslog 経由が受信し記録したログ行数が一致していることがわかる．UDP を用いていることから，パケットは送信順にログに記録されるわけではないが，パケットは取りこぼしなく受信している．

5.3 データの読み込みと画像生成

本研究では受信したデータから特定の 10 秒間を抜き出し秒ごとの PNG 画像を生成した後，GIF アニメーションを生成する．データはメモリ上で処理され，特定のデータ抜き出しに関しても Python の DataFrame を用いて処理される．PNG 画像作成に関しては create_graph 関数へデータが渡された後，引数として渡された unixtime を検索キーとして指定秒のデータのみが抜き出される．シーケンシャルに 10 秒分の画像を生成する処理に関しては，4.4 節で説明した処理フローとなるが，この処理を並列化し処理を高速化することができる．特に実験の環境は CPU コアが 8 個（HyperThreading 利用により 16 個）存在するため，コアスケールを考慮した実装が可能となる．

```
for i in range(10):
    t_array.append(t)
    t += 1

Parallel(n_jobs=10)([delayed(create_graph)(x)
    for x in t_array])

def create_graph(time):
    df2 = df[df.time == time]
    status = df2.status

    Z = status.values.reshape(100,100)

    ax0.imshow(Z, cmap=cmap)
    plt.savefig('./img/' + str(time) + '.png')
```

図 7　画像生成処理の並列化
Fig. 7　Parallel image generation

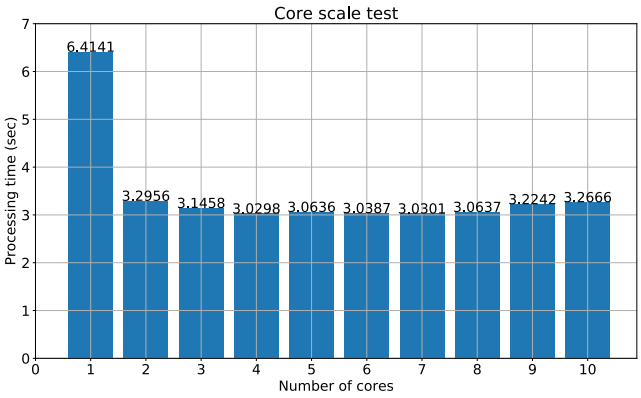

図 8　Core スケール評価
Fig. 8　CPU core scale evaluation

5.3.1 並列処理

画像の並列処理のために Python の Joblib ライブラリを用いた．これによりシーケンシャル処理であった画像生成部分を，図 7 で示すコードのように並列化することが可能となる．今回は 10 CPU コアを用いて並列化を行った．

並列処理による効果を測定した図が，図 8 である．この処理速度は，各 CPU コア数での処理時間の 10 回試行した平均時間となる．コア 1 つの場合と 2 つの場合では二倍近くの処理速度の差が確認できたが，2 コアから 10 コアの間

にはそこまで大きな処理速度の差はみられなかった．

6. 考察

6.1 UDP 受信チューニング

rsyslogd で 10,000pps の UDP パケットを受信しようとした場合に，送信したパケットが正しく受信できない事象を観測した．rsyslogd 自身のパフォーマンスチューニングを行なったが改善されず，netstat コマンドにてパケットドロップを確認したところ「receive buffer errors」の増加を確認した．これは OS のソケットバッファの値が足りず，UDP パケットをドロップしている状態であった．この事象を改善するために，/etc/sysctl.conf に以下を追記した．

```
net.core.rmem_default=8388608
net.core.rmem_max=8388608
```

これにより，UDP パケットのドロップは停止し，10,000pps の UDP パケットを rsyslogd が全て受信し，ファイルへと書き込むことを確認した．

6.2 全体の処理時間

本研究では，準リアルタイム処理としての可視化システムを実現した．ここでの準リアルタイムについての処理時間について述べる．

受信したデータは 1 ファイルに syslog 情報として記録されていく．このファイルは 1 分に一回 cron により，ログローテーションが行われる．ファイルをまたぐ時刻のグラフを生成するには，ローテーション前のログファイルとローテーション後のログファイルの 2 つのファイルを参照する必要がある．

graph generator の動作間隔をどの程度にするかにより，生成される画像数とその画像を基に作られる GIF アニメーションの長さが決まる．例えば，10 秒分のアニメーションを作成する場合，新旧二つのログファイルからデータを抜き出すことが求められる．graph generator は Pandas Dataframe を用いてデータを処理するため，読み込んだデータをソートした上で指定秒のデータを選択する．つまり，ハートビートパケットの受信遅延が発生した場合にも，新旧ファイルのどちらかにデータがある場合には，ログファイル自体がバッファリングのような機構となり画像生成時に遅延が吸収される．

graph generator をどの程度の間隔で動作させるか，また何秒の GIF アニメーションを生成させるかは，動作させる環境の CPU クロック数，コア数，メモリ容量，ディスクへのアクセス速度に応じて最大値，最小値が決定し，これらの値が事実上のチューニングパラメータとなる．

本実験は，表 1 に示される環境で実施した．パラメータとして，5 秒間隔で graph generator を動作させ，10 秒の

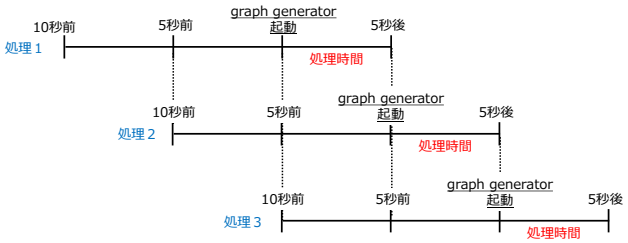

図 9 図の生成処理時間の概念
Fig. 9 Processing images of image generation

GIF アニメーションを生成させることに成功した．

図 9 に処理時間の概念を図示する．処理 1，処理 2，処理 3 は 5 秒おきに起動される graph generator を意味する．graph generator は特定時間をパラメータとして与えらる．graph generator に 5 秒前の時間を指定した場合，5 秒分の受信遅延を考慮し，指定時間の前後 5 秒を，つまり起動時間から逆算し前 10 秒分から起動時間までをデータ取得の範囲とする．そして，その 10 秒分のデータを用い，10 秒分の GIF アニメーションを生成する．これを 5 秒おきに繰り返すことにより連続した GIF アニメーションが生成される．10 秒分のデータを対象にしていることから，5 秒分の GIF アニメーションは前回 graph generator が生成した範囲と同時間の画像を生成する．こちらに関しては，5 秒分のバッファを超えてデータを遅延して受信した場合，再描画可能であるため無駄に見える重複処理にも意味はあるものと考える．

6.3 処理のスケーラビリティ

本研究では CPU コアスケールを目的として，処理の並列化を評価した．しかしながら 5.3.1 節では，CPU コアが 2 つ以上では処理速度に大きな差が見られなかった．コア数が 4 の時に平均処理速度が最速となり，コア数が 10 の時よりも早く処理が終了した．この処理遅延は Python の並列処理時に，メモリ上に展開された処理データを各 CPU コアへ処理プロセスを割り当てる時に行われるメモリコピーに関連する遅延が原因ではないかと推測する．適切な CPU コア数と適切なメモリコピーのタイミングで今回の環境ではコア数 4 が最速になったと考えられる．各環境に適したプログラムのパラメータのチューニングが必要である．

7. 今後の課題

本研究では 1 万という数値を基にターゲット監視の準リアルタイム処理と可視化を実装した．しかし，今後ターゲットの数がさらに増加していく可能性は大いに考えられ

ⓒ 2020 Information Processing Society of Japan

る．その場合に，1 つのシステムで 10 万ターゲット，100 万ターゲットとデータ処理行うことは，ハードウェア性能，ネットワーク帯域，ストレージ性能，その他計算機資源の限界を考慮すると性能のスケールアップには限界がある．そこで，Multi-access Edge Computing (MEC) [14] のように処理を分散化，効率化させ集約した結果を中央で処理するリソースへと集めることで，処理を分散させることにより処理の規模をスケールアウトすることが可能であると考える．

本研究ではマイクロサービスを含む大量のインスタンスがデプロイされた場合を想定し，それらのステータスを監視するための仕組みを例として実装した．本研究をベースとし，UDP パケットに情報を付加することでステータス以外の値を監視することができる．例えば付随情報に，Global Navigation Satellite System (GNSS) から取得可能な経度緯度の情報を付加することで，タイル状に表現していたステータス情報を，地図にマッピングすることで位置情報として扱うことが可能となる．これは大量の自動車が今どこを走っているかの可視化の様に，自動車の状態を準リアルタイムに表現する場合に有効に働くと考える．

参考文献

[1] Zabbix. https://www.zabbix.com/.
[2] Nagios. https://www.nagios.org/.
[3] Datadog. https://www.datadoghq.com/.
[4] Mackerel. https://mackerel.io/.
[5] 安田和磨, 石原真太郎, and 秋山豊和. 分散型 mqtt broker を活用したコンポーネント選択手法の比較評価. In **インターネットと運用技術シンポジウム論文集**, volume 2019, pages 25–32, nov 2019.
[6] Rrdtool. https://oss.oetiker.ch/rrdtool/.
[7] K. Hamada M. Matsuki H. Abe Y. Tsubouchi, A. Wakisaka and R. Matsumoto. Heterotsdb: An extensible time series database for automatically tiering onheterogeneous key-value stores. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Milwaukee, WI, USA*, pages 264–269, 2019.
[8] Hiroshi Abe, Keiichi Shima, Daisuke Miyamoto, Yuji Sekiya, Tomohiro Ishihara, Kazuya Okada, Ryo Nakamura, and Satoshi Matsuura. Distributed hayabusa: Scalable syslog search engine optimized for time-dimensional search. In *Proceedings of the Asian Internet Engineering Conference*, AINTEC '19, page 9–16, New York, NY, USA, 2019. Association for Computing Machinery.
[9] 雨宮宏一郎, 佐々木健吾, and 中尾彰宏. エッジコンピューティングを活用した広域分散処理システムにおける iomt サービス最適化のための metrics の提案 (ネットワークシステム). **電子情報通信学会技術研究報告 = *IEICE technical report :* 信学技報**, 117(33):41–46, may 2017.
[10] NETDATA. https://www.netdata.cloud/, 2007.
[11] R. Gerhards. The syslog protocol. RFC 5424, RFC Editor, March 2009. `http://www.rfc-editor.org/rfc/rfc5424.txt`.
[12] rsyslogd. https://www.rsyslog.com/.
[13] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.
[14] Hiroyuki Tanaka, Masahiro Yoshida, Koya Mori, and Noriyuki Takahashi. Multi-access edge computing: A survey. *Journal of Information Processing*, 26:87–97, 2018.

ⓒ 2020 Information Processing Society of Japan