hsvr 取扱説明書

libevent　ラッパーライブラリ

# はじめに

libeventは高速大容量なTCPアプリケーションの実装を可能にするオープンソースライブラリです。しかし多くの機能とAPI関数があり、使うには多くのドキュメントとサンプルプログラムの学習が必要です。

libeventを使うと高速なアプリケーションが実装可能になる理由の一つに、アプリケーションのソフトウェア構造が、ソケットプログラミングのAPI呼び出しアーキテクチャからイベントコールバック型に変更される点があります。

hsvrは、開発対象アプリケーションがTCP/TLSサーバであると想定して、イベントコールバック構造のアプリケーションでありながら機能を削減して、API数を削減して簡易で使いやすくするためのlibeventのラッパーライブラリです。しかし、従来のソケットインタフェース構造のまま、libeventと接続するミドルウェアではありません。アプリケーションはイベントが発生するごとに呼び出されるコールバック関数を処理するイベントドリブン型で記述する必要があります。

本ドキュメントは、hsvrの利用方法を以下の３つのパートに分けて解説します。

・libeventを含むhsvrの利用環境、およびアプリケーションビルド方法

・hsvrのライブラリ関数

・hsvrを使って実装した簡易HTTPサーバ例

# 環境およびビルド方法

hsvrはLinux上でTCP/TLSサーバアプリケーションを開発するためのlibeventラッパーライブラリです。libeventはLinuxに限らず多くの種類のOSで動作するオープンソースライブラリですが、hsvrはLinuxOS上で動作するC言語で記述されたミドルウェアです。たとえばpthreadライブラリを呼び出しますので、LinuxOSのみで使用できます。開発するサーバアプリケーションはOpenSSLとリンクしてTLSにも対応するために、ラップするlibeventのバーションは2.0.6-rc以上が必要となります。

開発マシンには、OpenSSLの開発者用パッケージとlibeventがインストールされている必要があります。libeventは主要なLinuxディストリビューションではlibevent-1がプリインストールされています。hsvrはlibevent-2を使いますので、libevent-1はアンイントールして、新規にlibevent-2をインストールしてください。

参考URL　 <https://github.com/downloads/libevent/libevent/libevent-2.0.21-stable.tar.gz>

OpenSSLとlibeventは共有ライブラリとしてインストールされています。　一方hsvrはいくつかのソースコードとして配布され、アプリケーションのビルド時に一緒にコンパイル・リンクします。

下記は、hsvr(hsvr.c およびstorage.c)をアプリケーションにリンクするためのMakefileの例です。hsvrはコンパイルしてオブジェクトファイルとしてアプリケーション(main.cとlog.c)とともにリンクします。OpenSSLとlibeventはリンカの-lオプションで取り込みます。

CFLAGS=-Wall -Wconversion

OBJ=log.o main.o storage.o hsvr.o

$(T): $(OBJ)

gcc -o $(T) $(OBJ) -lpthread -lssl -lcrypto -levent -levent\_openssl -levent\_pthreads

.c.o:

gcc $(CFLAGS) -c $< -ggdb

出来上がったアプリケーションを実行する場合に、OpenSSLとlibeventがダイナミックリンクをかけられます。OpenSSLは通常パッケージとしてインストールされていますが、libeventをソースコードからビルドしてインストールした場合には、LD\_LIBRARY\_PATHにインストール位置を指定する必要がありますのでご留意ください。

# ライブラリ関数

## TCPサーバ開始

int S\_start(uint16\_t port, int limit, int multi,CALLBACK\_T cb,int tls,char \*cert,char priv) ;

port:オープンするポート番号

limit:許容する同時最大コネクション数

multi:受信用スレッド数

cb:コールバック関数アドレス

tls:１＝TLSサーバ／0=TCPサーバ

cert:TLSの場合の証明書のパス

priv:TLSの場合の秘密鍵のパス

本関数はTCP/TLSサーバを開始します。サーバ動作中は、この関数から制御は戻りません。サーバをデーモン化する場合には、本関数を呼び出す前に子プロセスを生成してください。TLSサーバを開始する場合、内部でOpenSSLが初期化されてサーバ秘密鍵が読み込まれます。デーモン化する場合に、標準入出力を閉じてから本関数を呼び出すと、OpenSSLの表示する秘密鍵の暗号パスワードのプロンプトが表示されません。このために、privに指定する秘密鍵は事前に暗号化解除しておいてください。

以降、サーバに対してイベントが発生するとcbで指定した関数がコールバックされます。

## コールバック

S\_startを呼び出してサーバを開始すると、それ以降クライアントとの間に発生するTCP/TLS接続、データ受信、コネクション切断などの非同期に発生するイベントは、コールバック関数で通知されます。通知されるイベントの種類は、下記のCTYPE型で定義されています。

コールバックプロトタイプは、void型で第1パラメータにCTYPE、第２パラメータにSESSION構造体データへのアドレスをとります。アプリケーションは、この関数を実装してS\_startの第4パラメータにその開始アドレスを指定します。typeパラメータで発生したイベントの種別を、sessionパラメータでは該当のTCP/TLSコネクションに関する情報が格納されています。SESSION構造体の各メンバーは、usrを除き書き込み禁止です。

S\_startで制御がブロックすると、サーバではTCP/TLSコネクションの開設待ちになります。クライアントとのコネクション開設が行われると、Acceptコールバック（第１パラメータtypeにTYP＿ACCEPTが設定されたコールバック）が発生します。この時点でアプリケーションは新しくコネクションが開設されたことが検知できますので、アプリケーションとしてのセッション管理を開始します。具体的にはセッションに関わるデータを確保して、SESSION構造体のusrにセットしてコールバックをリターンします。

クライアントよりデータが受信されると、TYP\_READコールバックが発生します。このとき渡されるSESSION構造体のusrフィールドには、TYP\_ACCEPTイベント時に設定したセッション情報が引き継がれています。また、受信されたデータはbuffにセットされています。

クライアント側からセッション切断があった場合にはTYP\_TERMが発生します。これ以降、このコネクションに関わるコールバックは発生しません。TYP\_TERMが発生したら、usrフィールドに設定したセッション情報のメモリ領域はアプリケーション側で開放してください。

TYP\_TIMEOUTはTYP\_ACCEPTもしくはTYP\_READ以降にデータ受信がなかったコネクションに対して発生します。アプリケーション側で特に何もすることがなければ、そのままリターンしてください。タイマー時間はデフォルトで60秒です。この値はS\_settimerライブラリ関数で変更可能です。

typedef void(\*CALLBACK\_T)(CTYPE type,SESSION \*session);

typedef enum{

TYP\_ACCEPT,//新しいコネクションが開設された

TYP\_READ,//データが受信された

TYP\_TERM,//コネクションが切断された

TYP\_TIMEOUT,//受信タイムアウトが発生した

　　TYP\_SENDABLE//送信輻輳状態から送信可能状態への回復

}CTYPE;

typedef struct \_session\_t{

unsigned int id;

struct bufferevent \*bev;

struct bufferevent \*ubev;

int status;

size\_t txoctet;

size\_t tx;

size\_t rxoctet;

size\_t rx;

size\_t timeout;

　　int writewait;

unsigned char \*buff;

size\_t blen;

char host[32];

unsigned short port;

char \*usr;

}SESSION;

### TYP\_ACCEPT

クライアントとのTCPコネクションが開設され、TLSの場合にはネゴシエーションが成功すると、TYP\_ACCEPTコールバックが発生します。このコンテキストはS\_startを呼び出してブロックしたスレッドです。クライアント側のホスト情報はSESSION構造体のhostフィールドと　portフィールドに設定されています。S\_startで設定したコネクション上限を超えたコネクションの開設要求があると、そのコネクションは即時にサーバ側から閉じられ、グローバル変数のsession\_errorカウンタがインクリメントされます。この場合、コールバックは発生しません。

### TYP\_READ

クライアントからデータが受信されると、TYP\_READコールバックが発生します。hsvr内部では、S\_startにより指定された数のスレッドがワーカスレッドとして起動され、コネクション開設時にラウンドロビンでこのワーカスレッドのうちのいずれかに受信処理が割り当てられます。よってTYP\_READコールバックのコンテキストは、S\_startを呼び出したスレッドとは別のスレッドになっています。また、ワーカスレッドが複数個動作している場合には、アプリケーションがTYP\_READコールバックを実行中でも、別のワーカスレッドコンテキストでTYP\_READコールバックが発生します。このため、TYP\_READコールバックの処理関数は、再入可能なプログラムとしてください。

ひとつのTCPコネクションはひとつのスレッドに割り当てられ、コネクションが終了するまでそのスレッドが受信処理を受け持ちます。したがって、同一のTCPコネクションのSESSION情報をパラメータとして再入コールバックが発生することはありません。

受信されたデータはSESSION構造体のbuffフィールドポインタで示された開始アドレスからblenフィールドで示されるサイズが格納されています。一回のコールバックで、受信パケット１個が保証されているとは限りません。複数パケットがまとめて渡されることもあれば、ひとつのパケットの一部だけが渡されることもあり得ます。よってコールバック発生回数は、必ずしも受信パケット数と等しいわけではありません。一回のコールバックで渡される受信データの最大サイズは64000バイトです。

受信データが格納されているメモリ領域は、TYP\_READコールバックをリターンした時点で即時に解放されてしまいます。よって、TYP\_READコールバック関数リターン後に受信データ領域を参照するとメモリアクセスエラーとなりプロセスが異常終了する可能性があります。TYP\_READコールバック関数リターン後に受信データ領域を参照するには、アプリケーション側管理のメモリ領域に受信データの実体をコピーしてください。

### TYP\_TERM

コネクションが切断されると、該当のSESSION情報をパラメータとしてTYP\_TERMが発生します。このコールバック関数をリターンすると、SESSION情報領域は解放されますので、usrフィールドにアプリケーション側でセッション情報をセットした場合には、その領域を解放してください。SESSION情報領域に格納されているトラフィック統計情報はtxoctet（送信オクテット数）; tx（送信ライブラリ関数の呼び出し回数）; rxoctet（受信オクテット数）;rx（受信コールバック発生回数）;timeout（受信タイムアウト発生回数）は、本コールバック関数がリターンされた直後にグローバル変数に加算されます。コネクションごとのトラフィック統計が必要な場合には、TYP\_TERMコールバック関数内で、統計情報を取り込んでください。

### TYP\_TIMEOUT

一定時間経過してもデータを受信しなかったコネクションに関してはTYP\_TIMEOUTコールバックが発生します。タイマー時間はデフォルトで60秒です。この値はS\_settimerライブラリ関数で変更可能です。このイベントが発生してもコネクションは継続しています。さらに受信を継続する場合には何もしないでリターンしてください。アプリケーション側からコネクションを切断する場合には得られたSESSION構造体データを引数としてS\_closeを呼び出してください。この場合にも、usrフィールドの指す領域を解放しておいてください。

### TYP\_SENDABLE

クライアントに対してデータを送信する場合、送信バッファが溢れてパケットロスを起こす可能性があります。このために、送信関数S\_sendは送信バッファが輻輳中でデータが送信できなかったことを示すエラー値を返すことがあります。アプリケーションは、S\_sendの返り値がバッファ輻輳中であれば、送信バッファに空きができるまで待ってから再送のために再度S\_sendを呼び出さなくてはなりません。送信バッファに空きが出るタイミングは非同期イベントであるために、コールバックで通知されます。このコールバックを受け取ったら該当SESSIONに対してデータを再送してください。

## データ送信

int S\_send(SESSION \*s, unsigned char \*buff, size\_t len);

パラメータ

SESSION \*s 送信先SESSION情報

unsigned char \*buff,　送信データ開始アドレス

size\_t len　送信データ長

返り値

0：成功

-1：内部エラー

-2：パラメータエラー

-4：輻輳中エラー

本ライブラリ関数を使ってデータを送信します。送信先はコールバック、もしくはS\_searchにより得られたSESSION構造体を指定します。アプリケーションで独自に作成したSESSION構造体では内部エラーが返り送信できません。通常サーバアプリケーションは、クライアントからのデータ受信があってデータを送信します。よって、データ受信コールバックコンテキストでこの関数が呼ばれるでしょう。データを送ってきたクライアントに対して、応答を返す場合の送信先は、そのコールバック関数で得られたSESSION構造体をそのまま、本関数の第一パラメータに渡します。しかし、アプリケーションがHTTPプロキシやSIPプロキシなどのように、受信クライアントとは別のクライアントに対してデータを転送する場合には、通常送信先のIPアドレスとポート番号情報を持っているだけであり、当該クライアントのSESSION構造体は持っていません。そこで、S\_sendに先だってS\_search関数によりSESSION構造体を得て、それをS\_send関数のパラメータとして渡してください。

本関数は非同期です。すなわち、本関数を呼び出して0が返されたからといっても、データ送信が完了しているわけではありません。送信するべきデータはlibeventライブラリ内の送信バッファに追加され、送信処理が開始される前に即時に制御が戻されます。

送信バッファはコネクションごとに個別に存在します。輻輳制御発動閾値が設定されており、S\_send関数内部では、指定された送信先向けの送信バッファのバッファ使用量を検査します。これが発動閾値以下であれば、データはバッファに追加されて本関数は成功します。一方発動閾値を超えていれば、データはバッファに追加されずに輻輳中エラーが返ります。そして当該コネクションは輻輳制御モードに入ります。輻輳制御モードにあるコネクションは、SESSION構造体のwritewaitフィールドが１になっています。アプリケーションはしばらく時間が経過したのちに、そのデータを再び本関数で送信する必要がありますが、輻輳制御モードにあるコネクションに対してS\_sendを発行しても依然として輻輳中エラーが返されます。

本関数は非同期ですので、関数から制御が返された後に送信が開始され送信バッファ量が逓減してゆきます。そのうち、バッファ量が輻輳解除閾値を下回るとTYP\_SENDABLEコールバックが発生します。TYP\_SENDABLEが発生した時点では送信バッファは完全に０クリアされているわけではありませんが、輻輳制御モードを解除して送信可能状態に復帰していますので、そのTYP\_SENDABLEコールバックコンテキストでS\_sendを再発行してください。

初送のS\_sendと再送のS\_sendは同じスレッドコンテキストで呼び出されますが、コールバックタイミングが異なりますので、再送バッファが関数間で引き継げるように工夫してください。TYP\_READコールバックやTYP\_SENDABLEコールバックは再入可能でなければなりません。輻輳しているコネクションはひとつとは限りません。よって送信が失敗したデータを一時保存する再送バッファを静的領域に配置することは推奨されません。たとえば、再送バッファを動的に確保しておき、SESSION構造体のusrに指定したアプリケーションセッション情報が再送バッファアドレスを指すようにしておくと、TYPE\_SENDABLEコールバックは再入可能にできます。

現在のところ、輻輳制御発動閾値は1MB、輻輳解除閾値は128KBになっています。１MB以上のデータを送信すると、その直後のS\_sendで輻輳制御モードに入る可能性があります。１MB以上のデータを送信して、輻輳制御発動閾値を超えたとしてそのS\_send発行時点ではまだ輻輳制御発動閾値を超えていませんでしたから、輻輳制御モードに入っていません。すなわち、輻輳解除閾値を下回ってもTYPE\_SENDABLEコールバックは発生しません。

## SESSION構造体検索

SESSION\* S\_search(char \*ip,uint16\_t port);

パラメータ

char \*ip,　IPアドレス　例 “192.168.0.1”

uint16\_t　ポート番号

返り値

　SESSION構造体へのアドレス

該当コネクションが存在しない場合には、NULLが返されます。

## コネクションクローズ

void S\_close(SESSION \*s);

パラメータ

SESSION \*s 送信先SESSION情報

該当コネクションをクローズします。本関数は非同期であり、制御が戻ってもTCPコネクションが閉じられていることを保証するものではありません。特にS\_sendの直後にS\_closeを発行しても、送信バッファ内に未送信のデータが残っている場合には、即時にコネクションをクローズするのではなく、送信バッファ量が０になるのを待ってからクローズ処理が開始されます。これらは、ライブラリで行われる内部処理であり、アプリケーション側にはS\_closeからの制御は即時に返されています。

輻輳制御モードにあるコネクションに対してS\_closeを発行すると、やはり送信バッファが0になるまでクローズ処理は遅延しますが、その間に送信バッファ量が輻輳解除閾値を下回るタイミングに遭遇します。しかし、S\_closeが発行されたコネクションに対するTYPE\_SENDABLEコールバックは発生しません。コールバックが発生しないまま、送信バッファ内データが送信され終わったらコネクションが閉じされてSESSION構造体が解放されます。

## 受信タイムアウト値変更

void S\_settimer(SESSION \*s,int sec);

パラメータ

SESSION \*s SESSION情報

int sec　タイムアウト秒数

タイムアウトは、前のイベント発生時刻からの経過時間が指定秒数を超えた時点で発生します。

# 統計情報

ライブラリはSIG\_INT,SIG\_TERMをキャッチします。したがって端末が接続されているプロセスでは、コントロールCキーでサーバを停止することができます。シグナルハンドラ内ではライブラリ内部の統計情報を標準出力に出力してからプロセスが終了します。

ライブラリ内部で保持している統計情報は以下の通りです。各情報は32ビット符号なし整数に格納されています。したがって最大値は４Gで、それを超えると０にラップラウンドします。サンプルプログラムは１MBのコンテンツ返送するHTTPサーバを同梱していますが、ApacheBenchなどで高負荷試験をすると約4000コネクションで、送信オクテットカウンタがラップしますので、ご注意ください。

session\_gauge=0 現在接続中のコネクション数

session\_high=100 これまでの同時接続数の最大数（瞬間最大）

accept\_counter=4300 これまでのaccept累計数

session\_error=0 コネクション確立失敗回数

rxoctet=356900 受信オクテット数

rxcount=4300 受信コールバック発生回数

txoctet=5475604 送信オクテット数

txcount=4300 S\_send関数発行回数

rxtimeout=0 受信タイムアウトコールバック発生回数

discarded\_octet=0 廃棄オクテット数

discarded\_packet=0 廃棄イベント数

connect\_error=0 使用せず

thread 0 session=1075　スレッド0番がこれまでに受け持ったコネクション数

thread 1 session=1075　スレッド１番がこれまでに受け持ったコネクション数

thread 2 session=1075　スレッド２番がこれまでに受け持ったコネクション数

thread 3 session=1075　スレッド３番がこれまでに受け持ったコネクション数

# アプリケーション例

同梱のmain.c関数は、hsvrライブラリを使った簡易HTTPサーバと、telnetエコーサーバの２つの機能を含んでいます。コンパイル時に-DHTTPオプションを付加することでHTTPサーバとして操作します。このオプションがなければtelnetエコーサーバになります。

main関数

S\_start関数を呼び出すだけです。この例では同時接続数1000で、４ワーカスレッドで動作するTCPサーバを起動しています。

callback関数

コールバック種別により、内部関数である　do\_XXXに処理を振り分けます。

do\_accept

特に何もすることはありませんが、試験のためにタイムアウト値を５秒に変更しています。

do\_read

データ受信時に、そのリクエストの内容に関わらずHTTPサーバでは１MBのバイナリコンテンツを持つHTTPレスポンスを返します。telnetエコーモードでは、先頭文字を’!’に書き換え、それ以降のデータはそのまま返送します。

do\_timeout

タイムアウトが発生すると該当コネクションを切断します。

動作例

４スレッドでHTTPサーバを動作させて、ApacheBenchで同時接続100コネクションを5,000回接続する試験を実施しています。

[hata@vn-dev01 libe]$ ab -n 5000 -c 100 http://127.0.0.1:12345/

This is ApacheBench, Version 2.3 <$Revision: 655654 $>

Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/

Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient)

Completed 500 requests

Completed 1000 requests

Completed 1500 requests

Completed 2000 requests

Completed 2500 requests

Completed 3000 requests

Completed 3500 requests

Completed 4000 requests

Completed 4500 requests

Completed 5000 requests

Finished 5000 requests

Server Software:

Server Hostname: 127.0.0.1

Server Port: 12345

Document Path: /

Document Length: 1000000 bytes

Concurrency Level: 100

Time taken for tests: 13.320 seconds

Complete requests: 5000

Failed requests: 0

Write errors: 0

Total transferred: 5000515000 bytes

HTML transferred: 5000000000 bytes

Requests per second: 375.39 [#/sec] (mean)

Time per request: 266.391 [ms] (mean)

Time per request: 2.664 [ms] (mean, across all concurrent requests)

Transfer rate: 366628.29 [Kbytes/sec] received

Connection Times (ms)

min mean[+/-sd] median max

Connect: 1 7 4.3 6 27

Processing: 151 259 40.5 253 532

Waiting: 1 8 15.6 5 147

Total: 156 266 42.9 259 554

Percentage of the requests served within a certain time (ms)

50% 259

66% 262

75% 264

80% 265

90% 270

95% 272

98% 450

99% 523

100% 554 (longest request)

サーバ側で、終了時の統計情報出力は以下の通りです。

[hata@vn-dev01 libe]$ ./hsvr

^CStartCaught an SIG

session\_gauge=0

session\_high=100

accept\_counter=5000

session\_error=0

rxoctet=415000

rxcount=5000

txoctet=705547704

txcount=5000

rxtimeout=0

discarded\_octet=0

discarded\_packet=0

connect\_error=0

thread 0 session=1250

thread 1 session=1250

thread 2 session=1250

thread 3 session=1250

event\_base\_dispatch exits

アプリケーションサンプルソースコード

1 /\* =========================================================

2 Name : main.c

3 Author : H.Hata

4 Version :

5 Copyright : OLT

6 Description : Ansi-style

7 ========================================================== \*/

8 #include <stdio.h>

9 #include <stdlib.h>

10 #include <signal.h>

11 #include <string.h>

12 #include <stdint.h>

13 #include <unistd.h>

14 #include "hsvr.h"

15 static void do\_accept(SESSION \*s)

16 {

17 S\_settimer(s,5);

18 }

19 static void response(SESSION \*s,int len)

20 {

21 char \*buff;

22 int sz=len+512;

23 int slen,hlen;

24 int ret;

25 buff=malloc((size\_t)sz);

26 memset(buff,0,(size\_t)sz);

27 sprintf(buff,

28 "HTTP/1.1 200 OK\r\n"

29 "Content-Length: %d\r\n"

30 "Connection: Close\r\n"

31 "Content-Type: application/octet-stream\r\n"

32 "\r\n",len);

33 hlen=(int)strlen(buff);

34 slen=hlen+len;

35 s=S\_search(s->host,s->port);

36 ret=S\_send(s,(unsigned char \*)buff,(size\_t)slen);

37 free(buff);

38 S\_close(s);

39 if(ret!=0)

40 printf("ret=%d\n",ret);

41 }

42 static void do\_read(SESSION \*s)

43 {

44 #ifdef HTTP

45 if(s->blen>50) response(s,1000000);

46 #else

47 printf("%s\n",s->buff);

48 if(s->blen>0){

49 s->buff[0]='!';

50 S\_send(s,s->buff,s->blen);

51 }

52 #endif

53 }

54 static void do\_timeout(SESSION \*s)

55 {

56 printf("TIMEOUT %d \n",s->id);

57 S\_close(s);

58 }

59 static void do\_term(SESSION \*s)

60 {

61 printf("TERM %d\n",s->id);

62 }

63 static void callback(CTYPE type,SESSION \*s)

64 {

65 switch(type){

66 case TYP\_ACCEPT:

67 do\_accept(s);

68 break;

69 case TYP\_READ:

70 do\_read(s);

71 break;

72 case TYP\_TIMEOUT:

73 do\_timeout(s);

74 break;

75 case TYP\_TERM:

76 do\_term(s);

77 break;

78 default:

79 break;

80 }

81 }

82 static void daemonize(void)

83 {

84 int ret;

85 //1回目

86 fclose(stdin);

87 fclose(stdout);

88 fclose(stderr);

89 ret=fork();

90 if(ret>0){

91 //親プロセス

92 exit(EXIT\_SUCCESS);

93 }else if(ret<0){

94 exit(1);

95 }

96 //2回目

97 ret=fork();

98 if(ret>0){

99 //親プロセス

100 exit(EXIT\_SUCCESS);

101 }else if(ret<0){

102 exit(1);

103 }

104 }

105 int main(int argc,char \*\*argv)

106 {

107 uint16\_t port=12345;

108 int daemon=0;

109 int multi=4;

110 int limit=1000;

111 int tls=0;

112 char \*priv="pem/server.key";

113 char \*cert="pem/vng1\_256.crt";

114 printf("Start");

115 if(daemon==1){

116 daemonize();

117 }

118 S\_start(port,limit,multi,callback,tls,cert,priv);

119 return EXIT\_SUCCESS;

120 }