Async Sockets Documentation

Release 0.4.x

Efimov Evgenij

Apr 19, 2017

Contents

CHAPTER 1

Contents

Getting started

Installation

The recommended way to install async sockets library is through composer

stable version:

```
$ composer require edefimov/async-sockets:~0.3.0 --prefer-dist|--prefer-source
```

actual version:

```
$ composer require edefimov/async-sockets:dev-master
```

Use *–prefer-dist* option in production environment, so as it ignores downloading of test and demo files, and *–prefer-source* option for development. Development version includes both test and demo files.

Quick start

1. Create `AsyncSocketFactory` at the point where you want to start request. This object is the entry point to the library:

```php
use AsyncSockets\Socket\AsyncSocketFactory;

$factory = new AsyncSocketFactory();
```

2. Create *RequestExecutor* and proper amount of sockets:

```php
$client        = $factory->createSocket(AsyncSocketFactory::SOCKET_CLIENT);
$anotherClient = $factory->createSocket(AsyncSocketFactory::SOCKET_CLIENT);

$executor = $factory->createRequestExecutor();
```

3. Create event handler with events, you are interested in:

```php
use AsyncSockets\RequestExecutor\CallbackEventHandler;

$handler = new CallbackEventHandler(
    [
        EventType::INITIALIZE   => [$this, 'onInitialize'],
        EventType::CONNECTED    => [$this, 'onConnected'],
        EventType::WRITE        => [$this, 'onWrite'],
        EventType::READ         => [$this, 'onRead'],
        EventType::ACCEPT       => [$this, 'onAccept'],
        EventType::DATA_ALERT   => [$this, 'onDataAlert'],
        EventType::DISCONNECTED => [$this, 'onDisconnected'],
        EventType::FINALIZE     => [$this, 'onFinalize'],
        EventType::EXCEPTION    => [$this, 'onException'],
        EventType::TIMEOUT      => [$this, 'onTimeout'],
    ]
);
```

4. Add sockets into *RequestExecutor*:

```php
use AsyncSockets\Operation\WriteOperation;

$executor->socketBag()->addSocket(
    $client,
    new WriteOperation(),
    [
        RequestExecutorInterface::META_ADDRESS            => 'tls://example.com:443',
        RequestExecutorInterface::META_CONNECTION_TIMEOUT => 30,
        RequestExecutorInterface::META_IO_TIMEOUT         => 5,
    ],
    $handler
);
$executor->socketBag()->addSocket(
    $anotherClient,
    new WriteOperation(),
    [
        RequestExecutorInterface::META_ADDRESS            => 'tls://example.net:443',
        RequestExecutorInterface::META_CONNECTION_TIMEOUT => 10,
        RequestExecutorInterface::META_IO_TIMEOUT         => 2,
    ],
    $handler
);
```

5. Execute it!

```php
$executor->executeRequest();
```

The whole example may look like this:

```php
namespace Demo;

use AsyncSockets\Event\Event;
use AsyncSockets\Event\EventType;
use AsyncSockets\Event\ReadEvent;
use AsyncSockets\Event\SocketExceptionEvent;
use AsyncSockets\Event\WriteEvent;
```

```php
use AsyncSockets\Frame\MarkerFramePicker;
use AsyncSockets\RequestExecutor\CallbackEventHandler;
use AsyncSockets\RequestExecutor\RequestExecutorInterface;
use AsyncSockets\Operation\WriteOperation;
use AsyncSockets\Socket\AsyncSocketFactory;
use AsyncSockets\Socket\SocketInterface;

class RequestExecutorExample
{
    public function run()
    {
        $factory = new AsyncSocketFactory();

        $client        = $factory->createSocket(AsyncSocketFactory::SOCKET_CLIENT);
        $anotherClient = $factory->createSocket(AsyncSocketFactory::SOCKET_CLIENT);

        $executor->socketBag()->addSocket(
             $client,
             new WriteOperation("GET / HTTP/1.1\nHost: example.com\n\n"),
             [
                 RequestExecutorInterface::META_ADDRESS => 'tls://example.com:443',
             ]
        );

        $executor->socketBag()->addSocket(
            $anotherClient,
            new WriteOperation("GET / HTTP/1.1\nHost: example.net\n\n"),
            [
                RequestExecutorInterface::META_ADDRESS => 'tls://example.net:443',
            ]
        );

        $executor->withEventHandler(
            new CallbackEventHandler(
                [
                    EventType::WRITE      => [$this, 'onWrite'],
                    EventType::READ       => [$this, 'onRead'],
                ]
            )
        );

        $executor->executeRequest();
    }

    public function onWrite(WriteEvent $event)
    {
         $event->nextIsRead(new MarkerFramePicker(null, '</html>', false));
    }

    public function onRead(ReadEvent $event)
     {
         $socket   = $event->getSocket();
         $meta     = $event->getExecutor()->socketBag()->getSocketMetaData($event->
↪getSocket());

         $response = $event->getFrame()->getData();

         echo "<info>{$meta[RequestExecutorInterface::META_ADDRESS]}  read " .
```

```
                number_format(strlen($response), 0, ',', ' ') . ' bytes</info>';
    }
}
```

Here you create two sockets, the first will receive the main page from *example.net* and the second will receive mainpage from *example.com*. You should also inform the execution engine about the first I/O operation on the socket and destination address. These are minimum settings required for executing any request.

When connection is successfully established, since the `WriteOperation` is set, the *onWrite* method will be called by engine. Within write handler you tell the engine to prepare *read* operation with *marker* frame boundary detection strategy.

When the data are downloaded and is satisfied by given strategy, the *onRead* handler will be invoked, where you have access to downloaded data and some additional information about data.

Since in the *onRead* handler you don't ask the engine to prepare another I/O operation, the connection will be automatically closed for you.

The execution engine

AsyncSocketFactory

The `AsyncSocketFactory` is an entry point to the Async Socket Library. The factory is used to create sockets and request executors. You can use direct instantiation for this object:

```
use AsyncSockets\Socket\AsyncSocketFactory;

$factory = new AsyncSocketFactory();
```

Factory can be configured using `Configuration` object. The above code is the shortcut for configuring factory with default values:

```
use AsyncSockets\Socket\AsyncSocketFactory;
use AsyncSockets\Configuration\Configuration;

$configuration = new Configuration(
    [
        'connectTimeout'   => ini_get('default_socket_timeout'),
        'ioTimeout'        => ini_get('default_socket_timeout'),
        'preferredEngines' => ['libevent', 'native'],
    ]
);

$factory  = new AsyncSocketFactory($configuration);
```

Note: To see all configuration options see *Options reference*

Request executor

Request executor is a primary execution engine for all socket requests. Each time you need to run a request, you will use Request executor. Request executors are defined by `RequestExecutorInterface` which allows to

customize operations processing. There are two different implementations out of the box: native and libevent. The former is the pure php handler, the latter is based on libevent php library.

You can receive an instance of `RequestExecutorInterface` using the factory:

```
use AsyncSockets\Socket\AsyncSocketFactory;

$factory  = new AsyncSocketFactory();
$executor = $factory->createRequestExecutor();
```

The purposes of Request Executor are:

- Providing a bag for adding sockets (See *Setting up a socket* section below);
- Dispatching events during sockets' lifecycle;
- Executing request.

A request executor can be set up with global event handler, which will be applied to each added socket, and with limitation solver - an object restricting amount of executing requests at time.

Global event handler is the implementation of `EventHandlerInterface`, which will be called for every event on every added socket. There are four implementations of this interface out of the box:

- `CallbackEventHandler` takes array of callable, indexed by event type. For certain event type a certain callable will be invoked. Several callbacks can be defined for one event type;

```
$handler = new CallbackEventHandler(
          [
              EventType::INITIALIZE => [$this, 'logEvent'],
              EventType::WRITE      => [ [$this, 'logEvent'], [$this, 'onWrite']␣
↪],
              EventType::READ       => [ [$this, 'logEvent'], [$this, 'onRead'] ],
              EventType::EXCEPTION  => [$this, 'onException'],
              EventType::TIMEOUT    => [
                  [$this, 'onTimeout'],
                  function () {
                      echo "Timeout occured!\n";
                  }
              ],
          ]
);
```

- `EventHandlerFromSymfonyEventDispatcher` dispatches all socket event to symfony EventDispatcher;
- `EventMultiHandler` is the composite for `EventHandlerInterface` implementations;
- `RemoveFinishedSocketsEventHandler` decorator for any implementation of `EventHandlerInterface` which automatically removes completed sockets from `RequestExecutor`. Recommended to use for accepted clients from server sockets.

Note: You can register several global event handlers using `withEventHandler` method of `RequestExecutorInterface`.

The limitation solver is the component restricting amount of executed at once requests. Out of the box two strategies are available:

- `NoLimitationSolver` doesn't restrict anything, it is a default one;

- `ConstantLimitationSolver` restricts amount of running requests to given number.

Note: You can write custom limitation solver, see *Custom limitation solver*

To set up event handler or limitation solver use the following code:

```
$executor->withEventHandler(
     new CallbackEventHandler(
         [
             EventType::INITIALIZE => [$this, 'onInitialize'],
             EventType::WRITE      => [$this, 'onWrite'],
             ....
         ]
     )
);

$executor->withLimitationSolver(new ConstantLimitationSolver(20));
```

Socket lifecycle

During request socket pass through lifecycle shown in the figure below.

Each state except *added* and *removed* calls event handler with some information about occurred event.

Setting up a socket

Socket can be added into execution engine using `socketBag()` method from `RequestExecutorInterface`. It returns object of class `SocketBagInterface` allows to manage sockets. Socket bag is a container for all sockets processed by the engine. Every socket can have it's own event handler and options.

You can use the following code to add socket into *RequestExecutor*:

```
$executor->socketBag()->addSocket(
    $socket,
    new WriteOperation('some data'),
    [
        RequestExecutorInterface::META_ADDRESS            => 'tls://example.com:443',
        RequestExecutorInterface::META_CONNECTION_TIMEOUT => 30,
        RequestExecutorInterface::META_IO_TIMEOUT         => 5,
    ],
    $handler
);
```

Method `addSocket()` accepts four arguments: socket, operation, metadata and event handler. Socket is the object, created by `AsyncSocketFactory` or received by *AcceptEvent*. *Metadata* is a key-value array with settings for this socket. Event handler is an implementation of `EventHandlerInterface`, which will be invoked only for this socket.

Once you set up all sockets, you can execute the request:

```
$executor->executeRequest();
```

Warning: You should not create nested *RequestExecutor* during request processing.

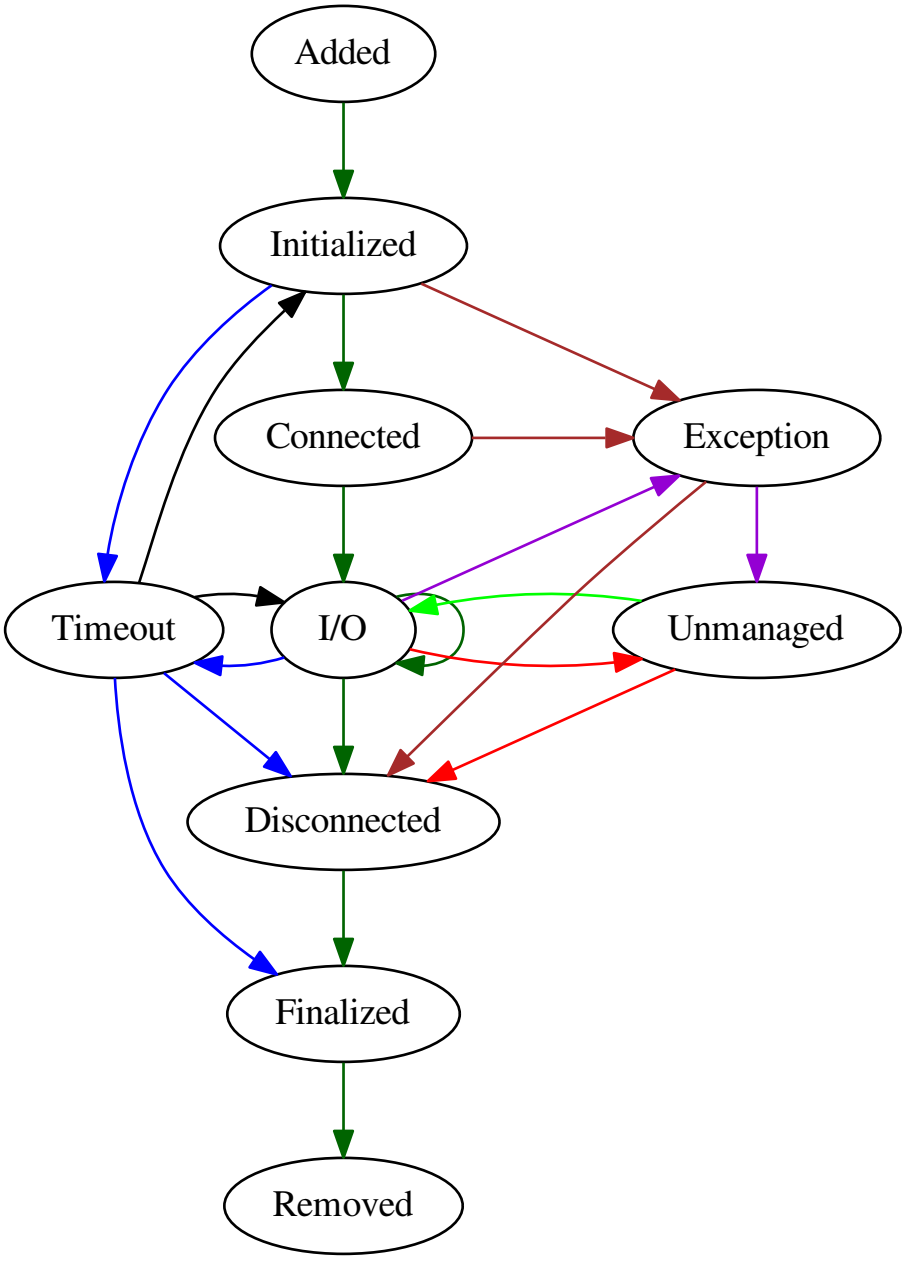

Fig. 1.1: Socket lifecycle

Socket types

Async Socket Library provides two types of sockets - client and server. The client socket can be either persistent or non-persistent. The recommended way of creating sockets of different types in source code is by using `AsyncSocketFactory`:

```
use AsyncSockets\Socket\AsyncSocketFactory;

$factory = new AsyncSocketFactory();

$client  = $factory->createSocket(AsyncSocketFactory::SOCKET_CLIENT);
$server  = $factory->createSocket(AsyncSocketFactory::SOCKET_SERVER);
```

You may pass additional options via second argument of `createSocket()`:

```
use AsyncSockets\Socket\AsyncSocketFactory;

$factory = new AsyncSocketFactory();
$client  = $factory->createSocket(
    AsyncSocketFactory::SOCKET_CLIENT,
    [
        AsyncSocketFactory::SOCKET_OPTION_IS_PERSISTENT => true
    ]
);
```

The above code will create persistent socket. See *persistent sockets* to get detailed information.

All available options are now linked with *persistent sockets*.

Processing data

The *socket lifecycle* is managed via operations.

Every action on the socket is described by an operation. Operations are implementations of `OperationInterface`. Each operation is an object, containing concrete data, required for processing. There are 5 available operations:

- `ReadOperation`
- `WriteOperation`
- `SslHandshakeOperation`
- `DelayedOperation`
- `NullOperation`

Note: For now there is no possibility to add custom operation into library.

ReadOperation

You can read data from socket using `ReadOperation`.

```
use AsyncSockets\Operation\ReadOperation;

$operation = new ReadOperation();
```

This code will create read operation telling the executing engine to handle reading data from the socket. By default every read operation will immediately call event handler with the data received from socket.

Note: Be ready to process empty data, if you are using `ReadOperation` without constructor arguments.

The default behaviour can be easily changed using its constructor argument accepting instance of `FramePickerInterface`:

```
use AsyncSockets\Operation\ReadOperation;
use AsyncSockets\Frame\FixedLengthFramePicker;

$operation = new ReadOperation(new FixedLengthFramePicker(50));
```

This code will emit *read event* when frame will be finished, i.e. in this case the 50 bytes of response will be received. See detailed explanations about *frame pickers*.

If given frame can not be received, the *exception event* is dispatched with `FrameException` object inside event argument.

If the remote site does not send any data within chosen period of time, the *timeout event* will be dispatched.

WriteOperation

The `WriteOperation` allows to send data to remote side. The data must be *string*. Each write operation will either send the whole given string without emitting any event or fail with some exception. The `WriteOperation` dispatches *write event* *before* sending the data.

```
use AsyncSockets\Operation\WriteOperation;

$executor->socketBag()->addSocket(
     $client,
     new WriteOperation("GET / HTTP/1.1\nHost: example.com\n\n"),
     [
         RequestExecutorInterface::META_ADDRESS => 'tls://example.com:443',
     ]
);
```

The example above will send the simplest GET request to example.com. If the remote site does not accept any data within chosen period of time, the *timeout event* will be dispatched.

SslHandshakeOperation

Normally when you intend to establish secured connection with remote host you use address like *tls://example.com:443* and it works perfect. With one great disadvantage - connection will be done synchronously even if you have switched socket into non-blocking mode. This happens because of SSL handshake procedure required for successful data exchange.

The `SslHandshakeOperation` allows to process the handshake asynchronously leaving the CPU time for some useful work.

Supposing you have request executor instance and socket created, you can connect to remote server asynchronously:

```
use AsyncSockets\Operation\SslHandshakeOperation;

$executor->socketBag()->addSocket(
```

```
    $socket,
    new SslHandshakeOperation(
        new WriteOperation("GET / HTTP/1.1\nHost: example.com\n\n")
    ),
    [
        RequestExecutorInterface::META_ADDRESS => 'tcp://example.com:443',
    ]
);
```

The `SslHandshakeOperation`'s constructor accept two arguments:

- the operation to execute after the socket has connected;
- the cipher to use for SSL connection, one of php constant *STREAM_CRYPTO_METHOD_*_CLIENT* for client sockets.

If the second parameter is omitted, the default value *STREAM_CRYPTO_METHOD_TLS_CLIENT* will be used.

If connection can not be established, the *exception event* is dispatched with `SslHandshakeException` object inside event argument.

> **Warning:** Do not use `SslHandshakeOperation` more than once for any socket request as the second call will fail with `SslHandshakeException`.

DelayedOperation

The `DelayedOperation` allows to postpone operation to some future time determined by a callback function. The callback function must answer the question *"Is an operation is still pending?"* and return *true* if socket is waiting for something and *false* when it is ready to proceed. The function is executed each time there is some *other* socket in the engine to process.

This feature is useful when a socket is waiting data from another one.

The constructor of `DelayedOperation` accepts three arguments:

- the operation to execute after delay is finished;
- the callback function;
- optional arguments to pass into callback function.

The callback function prototype must be the following:

```
bool function(SocketInterface $socket, RequestExecutorInterface $executor, ...
↪$arguments)
```

> **Warning:** The callback function is executed only when there is at least one socket except waiting one and there is some activity on the second socket. If these conditions are not met, the operation on the waiting socket will never finish.

NullOperation

The `NullOperation` is a special type of operation which is automatically set for socket, if the next operation has not been defined in *read event* or *write event*. This operation does not perform any action and has different meanings for persistent socket and non-persistent ones.

For non-persistent sockets `NullOperation` is considered as the end of request and the engine closes the connection.

For persistent sockets the situation significantly changes since persistent sockets keep connection all the time. If there are new data to read and `NullOperation` is set for the socket, the system dispatches *data alert event*. In the response to the event you can set the appropriate read operation and receive the data or close the connection.

Warning: If you don't do anything the connection will be automatically closed by the engine after some unsuccessful event calls and `UnmanagedSocketException` will be thrown.

How to work with frames

One of the key features of Async Socket Library is the determination of frame boundaries according to user-provided provided information.

FramePickers

FramePickers are the special objects implementing `FramePickerInterface` and designed to split incoming data into *frames*. FramePickers are passed into `ReadOperation` as the first argument of its constructor.

```
use AsyncSockets\Frame\FixedLengthFramePicker;

$picker    = new ...;
$operation = new ReadOperation($picker);
```

There are some implementations available out of the box.

- FixedLengthFramePicker
- MarkerFramePicker
- RawFramePicker
- EmptyFramePicker

Warning: *FramePickers*' instances are not reusable. Once you've set it into an operation, you can not reuse them in another one. Moreover you can not reuse the *FramePicker* after receiving *read event*. Create new instance each time instead.

Each *FramePicker* describes some rules that data should match before it can create a frame. If rule can not be met a `FrameException` is thrown in *exception event*.

The core idea of using *FramePickers* is that client code is aware of data structure it intends to receive from remote side.

FixedLengthFramePicker

The `FixedLengthFramePicker` allows to receive frame of well-known size. The only argument of its constructor accepts length of frame in bytes.

```
use AsyncSockets\Frame\FixedLengthFramePicker;

$operation = new ReadOperation(
    new FixedLengthFramePicker(255)
);
```

By setting this operation the *read event* will fired only after loading 255 bytes from remote side.

Note: Actually more data from network can be collected, but event will be fired with exactly given bytes of data.

MarkerFramePicker

If data have variable length, but there are well-known ending and beginning of data (or at least ending) it is possible to use `MarkerFramePicker`. It cuts the data between given start marker and end marker, including markers themselves.

Example usages:

```
use AsyncSockets\Frame\MarkerFramePicker;

$picker = new MarkerFramePicker('HTTP', "\r\n\r\n"); // return HTTP headers

$picker = new MarkerFramePicker(null, "\x00"); // reads everything until 0-byte␣
↪including 0-byte itself

$picker = new MarkerFramePicker("\x00", "\x00"); // start and end marker can be the␣
↪same

$picker = new MarkerFramePicker('<start>', '</START>', true); // returns everything␣
↪between <start> and </START>
                                                              // case-insensitive␣
↪compare
```

Warning: When you use a `MarkerFramePicker` and there are some data before the start marker passed into *FramePicker*, all these data will be lost. Suppose you have such incoming data:

AAA	X1234567890X	CCC

and such a *FramePicker* used for the first read operation:

```
$picker = new MarkerFramePicker("X", "X");
```

Since it is the first read operation, the data *AAA* will be lost.

RawFramePicker

This kind of *FramePicker* is used by default in `ReadOperation` if no other object is provided. With `RawFramePicker` the *read event* will be dispatched each time the socket read data.

Note: Be ready to process even an empty string using this *FramePicker*.

EmptyFramePicker

This *FramePicker* does not really read anything and the empty string is the always data for this frame. This frame has special meaning in SSL context for persistent socket - if there are some data in socket buffer which can not be treated as a frame, the *FramePicker* can clean it and stop the receiving of *data alert event*. This kind of garbage collection can be done automatically by decorating your event handler into `SslDataFlushEventHandler`.

Frames

After `FramePicker` has finished reading data it can create a *Frame*. Frames are a piece of data collected by current read operation. Each frame implements `FrameInterface` providing methods for getting the data and remote host address these data belongs to.

```
public function onRead(ReadEvent $event)
{
    $remoteAddress = $event->getFrame()->getRemoteAddress();
    echo "Received data from {$remoteAddress}: \n\n" . $event->getFrame()->getData();
}
```

An alternative way of receiving data from the frame is casting an object into a string:

```
public function onRead(ReadEvent $event)
{
    $remoteAddress = $event->getFrame()->getRemoteAddress();
    echo "Received data from {$remoteAddress}: \n\n {$event->getFrame()}";
}
```

Out of the box 3 implementations of Frames are available:

- `Frame` - default implementation for creating finished piece of data;
- `PartialFrame` - an implementation showing that data are incomplete. You will never receive it in read event, but if you intend to create a custom `FramePicker`, you should use this type of frame as a result of `createFrame()` method from `FramePickerInterface` when the system calls it;
- `EmptyFrame` - a frame that always casted into an empty string. This object is returned by `EmptyFramePicker`.

Limitation solver

The LimitationSolver is the component which allows to restrict amount of executing requests at certain period of time.

Writing custom limitation solver

Custom limitation solver can be written by implementing `LimitationSolverInterface`.

The `LimitationSolverInterface` contains 3 methods:

- `initialize()` is called before engine execution loop is started;
- `finalize()` is called after engine execution loop is finished;
- `decide()` is called each time the engine needs to make some decision about the socket.

The prototype of decide method looks like:

```
public function decide(RequestExecutorInterface $executor, SocketInterface $socket,
↪$totalSockets);
```

The `decide` method should return a hint for engine what to do with certain given socket. The possible decisions are:

- DECISION_OK - schedule request for given socket;
- DECISION_PROCESS_SCHEDULED - the engine has enough scheduled sockets and should process them before taking new ones;
- DECISION_SKIP_CURRENT - this certain socket should not be processed right now.

If you need an access to socket events from the solver, just implement `EventHandlerInterface` in addition to the `LimitationSolverInterface` one.

The simple implementation of the `LimitationSolverInterface` is `ConstantLimitationSolver`:

```
class ConstantLimitationSolver implements LimitationSolverInterface,␣
↪EventHandlerInterface
{
    private $limit;
    private $activeRequests;

    public function __construct($limit)
    {
        $this->limit = $limit;
    }

    public function initialize(RequestExecutorInterface $executor)
    {
        $this->activeRequests = 0;
    }

    public function decide(RequestExecutorInterface $executor, SocketInterface
↪$socket, $totalSockets)
    {
        if ($this->activeRequests + 1 <= $this->limit) {
            return self::DECISION_OK;
        } else {
            return self::DECISION_PROCESS_SCHEDULED;
        }
    }

    public function invokeEvent(Event $event)
    {
        switch ($event->getTypes()) {
            case EventType::INITIALIZE:
                $this->activeRequests += 1;
```

```
                break;
            case EventType::FINALIZE:
                $this->activeRequests -= 1;
                break;
        }
    }

    public function finalize(RequestExecutorInterface $executor)
    {

    }
}
```

Persistent sockets

Persistent sockets allow to reuse opened connection for subsequent requests. For creating a persistent socket you should pass additional options into `createSocket()` method from `AsyncSocketFactory`:

```
use AsyncSockets\Socket\AsyncSocketFactory;

$factory = new AsyncSocketFactory();
$client  = $factory->createSocket(
    AsyncSocketFactory::SOCKET_CLIENT,
    [
        AsyncSocketFactory::SOCKET_OPTION_IS_PERSISTENT => true
    ]
);
```

There are two options allow to control persistent socket connections:

- AsyncSocketFactory::SOCKET_OPTION_IS_PERSISTENT - a **boolean** flag, must be explicitly set to **true** for persistent sockets;
- AsyncSocketFactory::SOCKET_OPTION_PERSISTENT_KEY - **string**, an optional name to store this connection. Changing this value for the same host and port opens multiple connections to the same server.

Note: All persistent sockets must be explicitly closed.

Warning: When you process read or write event and don't set next operation on the persistent socket, the one becomes unmanaged. This means next time there will be new network activity, you will receive *data alert event*.

Event reference

To deal with sockets you need to subscribe to events you are interested in. Each event type has `Event` object (or one of its children) as the callback argument. If you have symfony event dispatcher component installed , the Async Socket Library's `Event` object will be inherited from symfony *Event* object. All events are described in `EventType` class.

INITIALIZE

Summary The first event sent for each socket and can be used for some initializations, for ex. setting destination address for client socket and local address for server ones.

Constant in `EventType` *INITIALIZE*

Callback argument `Event`

CONNECTED

Summary Socket has been just connected to server.

Constant in `EventType` *CONNECTED*

Callback argument `Event`

ACCEPT

Summary Applicable only for server sockets, fires each time there is a new client, no matter what kind of transport is used *tcp*, *udp*, *unix* or something else. Client socket can be got from *AcceptEvent*.

Constant in `EventType` *INITIALIZE*

Callback argument `AcceptEvent`

READ

Summary New frame has arrived. The `Frame` data object can be extracted from event object passed to the callback function. Applicable only for client sockets.

Constant in `EventType` *READ*

Callback argument `ReadEvent`

WRITE

Summary Socket is ready to write data. New data must be passed to socket through event object.

Constant in `EventType` *WRITE*

Callback argument `WriteEvent`

DATA_ALERT

Summary Socket is in unmanaged state. Event is fired when there are new data in socket, but `ReadOperation` is not set. This event can be fired several times, the typical reaction should be either closing connection or setting appropriate `ReadOperation`. If none of this is done, connection will be automatically closed and `UnmanagedSocketException` will be thrown.

Constant in `EventType` *DATA_ALERT*

Callback argument `DataAlertEvent`

DISCONNECTED

Summary Connection to remote server is now closed. This event is not fired when socket hasn't connected.

Constant in `EventType` *DISCONNECTED*

Callback argument `Event`

FINALIZE

Summary Socket lifecycle is complete and one can be removed from the executing engine.

Constant in `EventType` *FINALIZE*

Callback argument `Event`

TIMEOUT

Summary Socket failed to connect/read/write data during set up period of time.

Constant in `EventType` *TIMEOUT*

Callback argument `TimeoutEvent`

EXCEPTION

Summary Some `NetworkSocketException` occurred and detailed information can be retrieved from `SocketExceptionEvent`.

Constant in `EventType` *EXCEPTION*

Callback argument `SocketExceptionEvent`

Socket metadata reference

Metadata are settings for all operations on given socket. Supported keys are defined in `RequestExecutorInterface`.

You can pass these options either during adding a socket into engine's bag:

```
use AsyncSockets\Operation\WriteOperation;

$executor->socketBag()->addSocket(
    $client,
    new WriteOperation(),
    [
        RequestExecutorInterface::META_ADDRESS            => 'tls://example.com:443',
        RequestExecutorInterface::META_CONNECTION_TIMEOUT => 30,
        RequestExecutorInterface::META_IO_TIMEOUT         => 5,
    ]
);
```

or you can set these settings later via `setSocketMetaData()` method.

META_ADDRESS

Data type: string

Read-only: no

Summary: Remote address in form scheme://target, destination address for client socket and local address for server ones. This value is required for manually created sockets and can be ignored for accepted ones.

META_CONNECTION_TIMEOUT

Data type: integer

Read-only: no

Summary: Value in seconds, if connection is not established during this time, socket will be closed automatically and *TIMEOUT* event will be fired. If value is omitted then value from `Configuration` will be used.

META_IO_TIMEOUT

Data type: double

Read-only: no

Summary: Value in seconds, if no data are sent/received during this time, socket will be closed automatically and *TIMEOUT* event will be fired. If value is omitted then value from `Configuration` will be used.

META_USER_CONTEXT

Data type: mixed

Read-only: no

Summary: Arbitrary user data. This field is not used anyhow by the engine.

META_SOCKET_STREAM_CONTEXT

Data type:

- array
- resource
- null

Read-only: no

Summary: Settings to set up in socket resource.

If value is a resource it must be a valid stream context created by stream_context_create PHP function.

If value is array, it must contain two nested keys: *options* and *params*, which will be passed into stream_context_create corresponding parameters.

If value is null, the default context returned by stream_context_get_default PHP function will be used.

META_REQUEST_COMPLETE

Data type: bool

Read-only: yes

Summary: Value indicating that execution for this request is finished. Socket with this flag set can be safely removed from engine's socket bag.

META_CONNECTION_START_TIME

Data type:

- double
- null

Read-only: yes

Summary: Timestamp value, int part is seconds and float is microseconds, indicates when connection process is started.

If connection has not started yet, the value is null.

META_CONNECTION_FINISH_TIME

Data type:

- double
- null

Read-only: yes

Summary: Timestamp value, int part is seconds and float is microseconds, indicates when connection process was finished.

If connection has not finished yet, the value is null.

META_LAST_IO_START_TIME

Data type:

- double
- null

Read-only: yes

Summary: Timestamp value, int part is seconds and float is microseconds, indicates when last I/O operation has started.

If there were no I/O operation, the value would be null.

META_BYTES_SENT

Data type:

- int

Read-only: yes

Summary: Amount of bytes sent via this socket.

META_BYTES_RECEIVED

Data type:

- int

Read-only: yes

Summary: Amount of bytes received via this socket.

Note: This value counts data handled by stream wrapper, i.e. decompressed and decrypted.

AsyncSocketFactory configuration reference

The `AsyncSocketFactory` can be configured using non-standard values. To pass these value into the factory use `Configuration` object:

```
use AsyncSockets\Socket\AsyncSocketFactory;
use AsyncSockets\Configuration\Configuration;

$options       = ...; // array with options to set
$configuration = new Configuration($options);
$factory  = new AsyncSocketFactory($configuration);
```

You should retrieve options from some source and pass it as key-value array into `Configuration` object.

List of options

connectTimeout

Data type: double

Default value: from *socket_default_timeout* php.ini directive

Summary: Default value for execution engine to wait connection establishment before considering it as timed out.

ioTimeout

Data type: double

Default value: from *socket_default_timeout* php.ini directive

Summary: Default value for execution engine to wait some I/O activity before considering connection as timed out.

Note: Too low timeout values may result in frequent timeouts on sockets.

preferredEngines

Data type: string[]

Default value: ['libevent', 'native']

Summary: Preferred order of execution engines to try to create by `createRequestExecutor()` method from `AsyncSocketFactory`. Only *native* and *libevent* values are possible inside the array.

> **Warning:** Incorrect configuration for *preferredEngines* option will lead to *InvalidArgumentException* is thrown when you create the Request executor.

Details: There are two possible implementations of executing engine - *native* and *libevent*. The *libevent* one requires libevent extension installed, whereas a *native* one can work without any additional requirements. See the comparative table below.

Engine	Pros and cons
native	1. Works without any additional requirements. 2. Supports persistent connections 3. By default supports up to 1024 concurrent connections and requires PHP recompilation to increase this number.
libevent	1. Requires libevent extension 2. All versions prior to 0.1.1 can not process persistent connections and fails with "fd argument must be either valid PHP stream or valid PHP socket resource" warning. 3. Version 0.1.1 is available only from sources. 4. Process more than 1024 concurrent connections.