使用Libevent编写高性能跨平台的非阻塞网络程序 i

**使用Libevent编写高性能跨平台的异步网络程序**

使用Libevent编写高性能跨平台的非阻塞网络程序 ii

**REVISION HISTORY**

|  |  |  |  |
| --- | --- | --- | --- |
| NUMBER | DATE | DESCRIPTION | NAME |
|  |  |  |  |

使用Libevent编写高性能跨平台的非阻塞网络程序 iii

**Contents**

|  |  |  |  |
| --- | --- | --- | --- |
| [1](#page8) | [About this document](#page8) | | 2 |
| [2 A note on examples](#page8) | | | 2 |
| [3 A tiny introduction to asynchronous IO](#page8) | | | 2 |
|  | [3.1](#page21) | [What about convenience? (and what about Windows?)](#page21) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 15 |
|  | [3.2](#page24) | [How efficient is all of this, really?](#page24) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 18 |
| [4](#page24) | [The Libevent Reference Manual: Preliminaries](#page24) | | 18 |
|  | [4.1](#page24) | [Libevent from 10,000 feet](#page24) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 18 |
|  | [4.2](#page25) | [The Libraries](#page25) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 19 |
|  | [4.3](#page26) | [The Headers](#page26) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 20 |
|  | [4.4](#page26) | [If you have to work with an old version of Libevent](#page26) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 20 |
|  |  | [4.4.1 Notes on version status](#page26) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 20 |
| [5 Setting up the Libevent library](#page27) | | | 21 |
|  | [5.1](#page27) | [Log messages in Libevent](#page27) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 21 |
|  | [5.2](#page28) | [Handling fatal errors](#page28) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 22 |
|  | [5.3](#page29) | [Memory management](#page29) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 23 |
|  | [5.4](#page30) | [Locks and threading](#page30) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 24 |
|  | [5.5](#page33) | [Debugging lock usage](#page33) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 27 |
|  | [5.6](#page33) | [Debugging event usage](#page33) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 27 |
|  | [5.7](#page35) | [Detecting the version of Libevent](#page35) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 29 |
|  | [5.8](#page36) | [Freeing global Libevent structures](#page36) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 30 |
| [6](#page37) | [Creating an event\_base](#page37) | | 31 |
|  | [6.1](#page37) | [Setting up a default event\_base](#page37) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 31 |
|  | [6.2](#page38) | [Setting up a complicated event\_base](#page38) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 32 |
|  | [6.3](#page40) | [Examining an event\_base’s backend method](#page40) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 34 |
|  | [6.4](#page41) | [Deallocating an event\_base](#page41) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 35 |
|  | [6.5](#page42) | [Setting priorities on an event\_base](#page42) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 36 |
|  | [6.6](#page42) | [Reinitializing an event\_base after fork()](#page42) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 36 |
|  | [6.7](#page43) | [Obsolete event\_base functions](#page43) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 37 |
| [7 Working with an event loop](#page43) | | | 37 |
|  | [7.1](#page43) | [Running the loop](#page43) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 37 |
|  | [7.2](#page44) | [Stopping the loop](#page44) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 38 |
|  | [7.3](#page46) | [Re-checking for events](#page46) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 40 |
|  | [7.4](#page46) | [Checking the internal time cache](#page46) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 40 |
|  | [7.5](#page46) | [Dumping the event\_base status](#page46) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 40 |
|  | [7.6](#page47) | [Running a function over every event in an event\_base](#page47) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 41 |
|  | [7.7](#page47) | [Obsolete event loop functions](#page47) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 41 |
|  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 使用Libevent编写高性能跨平台的非阻塞网络程序 | | | | iv |
| [8](#page47) | [Working with events](#page47) | | | 41 |
|  | [8.1](#page48) | [Constructing event objects](#page48) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 42 |
|  |  | [8.1.1](#page49) | [The event flags](#page49) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 43 |
|  |  | [8.1.2](#page50) | [About Event Persistence](#page50) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 44 |
|  |  | [8.1.3](#page50) | [Creating an event as its own callback argument](#page50) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 44 |
|  |  | [8.1.4](#page51) | [Timeout-only events](#page51) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 45 |
|  |  | [8.1.5](#page51) | [Constructing signal events](#page51) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 45 |
|  |  |  | [Caveats when working with signals](#page52) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 46 |
|  |  | [8.1.6](#page52) | [Setting up events without heap-allocation](#page52) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 46 |
|  | [8.2](#page54) | [Making events pending and non-pending](#page54) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 48 |
|  | [8.3](#page54) | [Events with priorities](#page54) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 48 |
|  | [8.4](#page55) | [Inspecting event status](#page55) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 49 |
|  | [8.5](#page56) | [Finding the currently running event](#page56) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 50 |
|  | [8.6](#page57) | [Configuring one-off events](#page57) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 51 |
|  | [8.7](#page57) | [Manually activating an event](#page57) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 51 |
|  | [8.8](#page58) | [Optimizing common timeouts](#page58) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 52 |
|  | [8.9](#page59) | [Telling a good event apart from cleared memory](#page59) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 53 |
|  | [8.10](#page60) | [Obsolete event manipulation functions](#page60) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 54 |
| [9](#page61) | [Helper functions and types for Libevent](#page61) | | | 55 |
|  | [9.1](#page61) | [Basic types](#page61) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 55 |
|  |  | [9.1.1](#page61) | [evutil\_socket\_t](#page61) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 55 |
|  |  | [9.1.2](#page62) | [Standard integer types](#page62) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 56 |
|  |  | [9.1.3](#page62) | [Miscellaneous compatibility types](#page62) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 56 |
|  | [9.2](#page62) | [Timer portability functions](#page62) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 56 |
|  | [9.3](#page63) | [Socket API compatibility](#page63) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 57 |
|  | [9.4](#page64) | [Portable string manipulation functions](#page64) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 58 |
|  | [9.5](#page65) | [Locale-independent string manipulation functions](#page65) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 59 |
|  | [9.6](#page65) | [IPv6 helper and portability functions](#page65) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 59 |
|  | [9.7](#page66) | [Structure macro portability functions](#page66) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 60 |
|  | [9.8](#page66) | [Secure random number generator](#page66) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 60 |
| [10](#page66) | [Bufferevents: concepts and basics](#page66) | | | 60 |
|  | [10.1](#page67) | [Bufferevents and evbuffers](#page67) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 61 |
|  | [10.2](#page67) | [Callbacks and watermarks](#page67) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 61 |
|  | [10.3](#page68) | [Deferred callbacks](#page68) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 62 |
|  | [10.4 Option flags for bufferevents](#page68) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | | 62 |
|  | [10.5 Working with socket-based bufferevents](#page69) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | | 63 |
|  |  | [10.5.1 Creating a socket-based bufferevent](#page69) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 63 |

|  |  |  |  |
| --- | --- | --- | --- |
| 使用Libevent编写高性能跨平台的非阻塞网络程序 | | | v |
|  | [10.5.2](#page69) | [Launching connections on socket-based bufferevents](#page69) . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 63 |
|  | [10.5.3](#page70) | [Launching connections by hostname](#page70) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 64 |
| [10.6](#page72) | [Generic bufferevent operations](#page72) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 66 |
|  | [10.6.1](#page72) | [Freeing a bufferevent](#page72) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 66 |
|  | [10.6.2](#page72) | [Manipulating callbacks, watermarks, and enabled operations](#page72) . . . . . . . . . . . . . . . . . . . . . . . . | 66 |
|  | [10.6.3](#page75) | [Manipulating data in a bufferevent](#page75) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 69 |
|  | [10.6.4](#page77) | [Read- and write timeouts](#page77) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 71 |
|  | [10.6.5](#page77) | [Initiating a flush on a bufferevent](#page77) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 71 |
| [10.7](#page77) | [Type-specific bufferevent functions](#page77) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 71 |
| [10.8](#page78) | [Manually locking and unlocking a bufferevent](#page78) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 72 |
| [10.9](#page78) | [Obsolete bufferevent functionality](#page78) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 72 |
| [11 Bufferevents: advanced topics](#page79) | | | 73 |
| [11.1](#page79) | [Paired bufferevents](#page79) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 73 |
| [11.2](#page80) | [Filtering bufferevents](#page80) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 74 |
| [11.3](#page81) | [Limiting maximum single read/write size](#page81) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 75 |
| [11.4](#page81) | [Bufferevents and Rate-limiting](#page81) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 75 |
|  | [11.4.1](#page82) | [The rate-limiting model](#page82) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 76 |
|  | [11.4.2](#page82) | [Setting a rate limit on a bufferevent](#page82) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 76 |
|  | [11.4.3](#page82) | [Setting a rate limit on a group of bufferevents](#page82) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 76 |
|  | [11.4.4](#page83) | [Inspecting current rate-limit values](#page83) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 77 |
|  | [11.4.5](#page83) | [Manually adjusting rate limits](#page83) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 77 |
|  | [11.4.6](#page84) | [Setting the smallest share possible in a rate-limited group](#page84) . . . . . . . . . . . . . . . . . . . . . . . . . . | 78 |
|  | [11.4.7](#page84) | [Limitations of the rate-limiting implementation](#page84) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 78 |
| [11.5](#page85) | [Bufferevents and SSL](#page85) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 79 |
|  | [11.5.1](#page85) | [Setting up and using an OpenSSL-based bufferevent](#page85) . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 79 |
|  | [11.5.2](#page88) | [Some notes on threading and OpenSSL](#page88) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 82 |
| [12 Evbuffers: utility functionality for buffered IO](#page89) | | | 83 |
| [12.1](#page90) | [Creating or freeing an evbuffer](#page90) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 84 |
| [12.2](#page90) | [Evbuffers and Thread-safety](#page90) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 84 |
| [12.3](#page90) | [Inspecting an evbuffer](#page90) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 84 |
| [12.4](#page90) | [Adding data to an evbuffer: basics](#page90) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 84 |
| [12.5](#page91) | [Moving data from one evbuffer to another](#page91) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 85 |
| [12.6](#page91) | [Adding data to the front of an evbuffer](#page91) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 85 |
| [12.7](#page92) | [Rearranging the internal layout of an evbuffer](#page92) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 86 |
| [12.8](#page92) | [Removing data from an evbuffer](#page92) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 86 |
| [12.9](#page93) | [Copying data out from an evbuffer](#page93) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 87 |
| [12.10Line-oriented input](#page94) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | | 88 |

|  |  |  |  |
| --- | --- | --- | --- |
| Fast portable non-blocking network programming with Libevent | | | vi |
| [12.11Searching within an evbuffer](#page95) . . . . . . . . . . . . . . . . . . . . . | | | . . . . . . . . . . . . . . . . . . . . . . . . 89 |
| [12.12Inspecting data without copying it](#page96) . . . . . . . . . . . . . . . . . . | | | . . . . . . . . . . . . . . . . . . . . . . . . 90 |
| [12.13Adding data to an evbuffer directly](#page98) . . . . . . . . . . . . . . . . . . | | | . . . . . . . . . . . . . . . . . . . . . . . . 92 |
| [12.14Network IO with evbuffers](#page100) . . . . . . . . . . . . . . . . . . . . . . | | | . . . . . . . . . . . . . . . . . . . . . . . . 94 |
| [12.15Evbuffers and callbacks](#page100) . . . . . . . . . . . . . . . . . . . . . . . . | | | . . . . . . . . . . . . . . . . . . . . . . . . 94 |
| [12.16Avoiding data copies with evbuffer-based IO](#page102) . . . . . . . . . . . . . | | | . . . . . . . . . . . . . . . . . . . . . . . . 96 |
| [12.17Adding a file to an evbuffer](#page103) . . . . . . . . . . . . . . . . . . . . . . | | | . . . . . . . . . . . . . . . . . . . . . . . . 97 |
| [12.18Fine-grained control with file segments](#page104) . . . . . . . . . . . . . . . | | | . . . . . . . . . . . . . . . . . . . . . . . . 98 |
| [12.19Adding an evbuffer to another by reference](#page105) . . . . . . . . . . . . . | | | . . . . . . . . . . . . . . . . . . . . . . . . 99 |
| [12.20Making an evbuffer add- or remove-only](#page105) . . . . . . . . . . . . . . . | | | . . . . . . . . . . . . . . . . . . . . . . . . 99 |
| [12.21Obsolete evbuffer functions](#page105) . . . . . . . . . . . . . . . . . . . . . . | | | . . . . . . . . . . . . . . . . . . . . . . . . 99 |
| [13 Connection listeners: accepting TCP connections](#page106) | | | 100 |
| [13.1](#page106) | [Creating or freeing an evconnlistener](#page106) . . . . . . . . . . . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 100 |
|  | [13.1.1](#page107) | [Recognized flags](#page107) . . . . . . . . . . . . . . . . . . . . . . . | . . . . . . . . . . . . . . . . . . . . . . . . 101 |
|  | [13.1.2 The connection listener callback](#page107) . . . . . . . . . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 101 |
| [13.2](#page108) | [Enabling and disabling an evconnlistener](#page108) . . . . . . . . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 102 |
| [13.3](#page108) | [Adjusting an evconnlistener’s callback](#page108) . . . . . . . . . . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 102 |
| [13.4](#page108) | [Inspecting an evconnlistener](#page108) . . . . . . . . . . . . . . . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 102 |
| [13.5](#page108) | [Detecting errors](#page108) . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 102 |
| [13.6](#page108) | [Example code: an echo server.](#page108) . . . . . . . . . . . . . . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 102 |
| [14 Using DNS with Libevent: high and low-level functionality](#page110) | | | 104 |
| [14.1](#page110) | [Preliminaries: Portable blocking name resolution](#page110) . . . . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 104 |
| [14.2](#page114) | [Non-blocking hostname resolution with evdns\_getaddrinfo()](#page114) . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 108 |
| [14.3](#page116) | [Creating and configuring an evdns\_base](#page116) . . . . . . . . . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 110 |
|  | [14.3.1 Initializing evdns from the system configuration](#page116) . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 110 |
|  |  | [The resolv.conf file format](#page117) . . . . . . . . . . . . . . . . . . | . . . . . . . . . . . . . . . . . . . . . . . . 111 |
|  | [14.3.2](#page118) | [Configuring evdns manually](#page118) . . . . . . . . . . . . . . . . . | . . . . . . . . . . . . . . . . . . . . . . . . 112 |
|  | [14.3.3](#page119) | [Library-side configuration](#page119) . . . . . . . . . . . . . . . . . . | . . . . . . . . . . . . . . . . . . . . . . . . 113 |
| [15 Low-level DNS interfaces](#page119) | | | 113 |
| [15.1](#page120) | [Suspending DNS client operations and changing nameservers](#page120) . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 114 |
| [16 DNS server interfaces](#page121) | | | 115 |
| [16.1](#page121) | [Creating and closing a DNS server](#page121) . . . . . . . . . . . . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 115 |
| [16.2](#page121) | [Examining a DNS request](#page121) . . . . . . . . . . . . . . . . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 115 |
| [16.3](#page122) | [Responding to DNS requests](#page122) . . . . . . . . . . . . . . . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 116 |
| [16.4](#page123) | [DNS Server example](#page123) . . . . . . . . . . . . . . . . . . . . . . . . . | | . . . . . . . . . . . . . . . . . . . . . . . . 117 |

[17 Obsolete DNS interfaces](#page125) 119

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 1 / 120 |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

These documents are Copyright (c) 2009-2012 by Nick Mathewson, and are made available under the Creative Commons Attribution-Noncommercial-Share Alike license, version 3.0. Future versions may be made available under a less restrictive license.

Additionally, the source code examples in these documents are also licensed

under the so-called "3-Clause" or "Modified" BSD license. See

link:license\_bsd.html[the license\_bsd file] distributed with these documents

for the full terms.

For the latest version of this document, see

http://www.wangafu.net/~nickm/libevent-book/TOC.html

To get the source for the latest version of this document, install git and run "git clone git://github.com/nmathewson/libevent-book.git" \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

This license applies to the example source code in the book, "Fast Portable non-blocking network programming in Libevent."

The book itself is NOT under this license.

Copyright 2009-2012 Nick Mathewson

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
* Neither the names of the copyright owners nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

These documents are Copyright (c) 2009-2012 by Nick Mathewson, and are made available under the Creative Commons Attribution-Noncommercial-Share Alike license, version 3.0. Future versions may be made available under a less restrictive license.

Additionally, the source code examples in these documents are also licensed under the so-called "3-Clause" or "Modified" BSD license. See [the license\_bsd file](license_bsd.html) distributed with these documents for the full terms.

For the latest version of this document, see <http://www.wangafu.net/~nickm/libevent-book/TOC.html>

|  |  |
| --- | --- |
| 使用Libevent编写高性能跨平台的非阻塞网络程序 | 2 / 120 |

To get the source for the latest version of this document, install git and run "git clone git://github.com/nmathewson/libevent-book.git"

**关于本文档**

本文档将教会你使用Libevent 2.0 (或更高版本)和C语言编写高性能跨平台的异步网络程序。 我们假设:

* 你已经熟悉C语言编程.
* 你了解基本的网络编程接口 (socket(),listen(),accept(), connect(),recv(),send()等等).

**A 示例代码小提醒**

本文档中的所有例子都在正确地运行在Linux, FreeBSD,OpenBSD, NetBSD, Mac OS X, Solaris, and Android. 部分示例可能不能在Windows上运行。

**异步网络编程小介绍**

非常多的初级程序员使用阻塞调用. 当你调用一个同步IO调用，它在IO完成时或超过了一定时间导致协议栈放弃才会返回。 例如：当你在tcp套接字上调用"connect()"时 ,你的操作系统发送了一个SYN 数据包到目标服务器. 操作系统将阻塞调用线程直到操作系统收到对端服务器的SYN ACK 数据包, 或者超过了很长时间操作系统决定放弃连接行为。

下面是一个简单的使用阻塞网络调用的客户端。 它连接上 www.google.com并且发送http请求, 将收到的返回输出到控制台.

Example: A simple blocking HTTP client

/\* For sockaddr\_in \*/

#include <netinet/in.h>

/\* For socket functions \*/

#include <sys/socket.h>

/\* For gethostbyname \*/

#include <netdb.h>

#include <unistd.h>

#include <string.h>

#include <stdio.h>

**int** main(**int** c, **char** \*\*v)

{

**const char** query[] =

"GET / HTTP/1.0\r\n"

|  |  |
| --- | --- |
| 使用Libevent编写高性能跨平台的非阻塞网络程序 | 3 / 120 |

"Host: www.google.com\r\n"

"\r\n";

**const char** hostname[] = "www.google.com";

**struct** sockaddr\_in sin;

**struct** hostent\*h;

**const char** \*cp;

**int** fd;

ssize\_t n\_written, remaining;

**char** buf[1024];

/\* 获取域名对应的IP地址. 注意：在多数操作系统中，这不是线程安全的. \*/

h = gethostbyname(hostname);

**if** (!h) {

fprintf(stderr, "Couldn’t lookup %s: %s", hostname, hstrerror(h\_errno)); **return** 1;

}

**if** (h->h\_addrtype != AF\_INET) {

fprintf(stderr, "No ipv6 support, sorry.");

**return** 1;

}

/\* 分配新 socket \*/

fd = socket(AF\_INET, SOCK\_STREAM, 0);

**if** (fd < 0) {

perror("socket");

**return** 1;

}

/\* 连接远程主机 \*/

sin.sin\_family = AF\_INET;

sin.sin\_port = htons(80);

sin.sin\_addr = \*(**struct** in\_addr\*)h->h\_addr;

**if** (connect(fd, (**struct** sockaddr\*) &sin, **sizeof**(sin))) {perror("connect");

close(fd);

**return** 1;

}

/\* 写请求. \*/

/\* XXX Can send succeed partially? \*/

cp = query;

remaining = strlen(query);

**while** (remaining) {

n\_written = send(fd, cp, remaining, 0);

**if** (n\_written <= 0) {

perror("send");

**return** 1;

}

remaining -= n\_written;

cp += n\_written;

}

/\* 获取返回数据. \*/

**while** (1) {

ssize\_t result = recv(fd, buf, **sizeof**(buf), 0);

**if** (result == 0) {

**break**;

} **else if** (result < 0) { perror("recv");

close(fd); **return** 1;

|  |  |
| --- | --- |
| 使用Libevent编写高性能跨平台的非阻塞网络程序 | 4 / 120 |

}

fwrite(buf, 1, result, stdout);

}

close(fd);

**return** 0;

}

在上面的程序中，所有网络调用都是阻塞的： gethostbyname()调用直到获取到www.google.com的IP地址或失败了才会返回; onnect()调用只有连接上远程主机才会返回; recv()直到收到数据或套接字关闭才会返回; send()直到数据复制到内核的写缓冲中才会返回。

阻塞调用也不是在所有情况下都不对。如果你的程序在阻塞时确实没有其它事情需要处理，阻塞调用可以工作得很好。但是如果你的程序在同一时间需要处理多个连接， 比如你需要读取两个连接的数据，你并不知道哪个连接的数据会先到。

错误做法示例：

/\* 这种做法将不会起作用 \*/

**char** buf[1024];

**int** i, n;

**while** (i\_still\_want\_to\_read()) {

**for** (i=0; i<n\_sockets; ++i) {

n = recv(fd[i], buf, **sizeof**(buf), 0);

**if** (n==0)

handle\_close(fd[i]);

**else if** (n<0)

handle\_error(fd[i], errno);

**else**

handle\_input(fd[i], buf, n);

}

}

如果在 fd[2] 的数据先到了, 你的程序在fd[0]和fd[1]读到数据前，不能收到fd[2]的数据。有时间大家用多线程或多进程来解决这个问题。一种最简单的多线程模型是为每个连接分配一个进程（或线程）。因为每个连接都有自己的进程，一个阻塞的调用只会阻塞它自己的进程，不会阻塞其它处理其它连接的进程。

下面是另一个程序示例。它监听40714端口，每次读取一行数据采用ROT13 编码后返回给客户端.它使用Unix 的fork()调用为每条连接创建一个进程。

Example: Forking ROT13 server

/\* For sockaddr\_in \*/

#include <netinet/in.h>

/\* For socket functions \*/

#include <sys/socket.h>

#include <unistd.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#define MAX\_LINE 16384

**char**

rot13\_char(**char** c)

{

/\* We don’t want to use isalpha here; setting the locale would change

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 5 / 120 |

* which characters are considered alphabetical. \*/ **if** ((c >= ’a’ && c <= ’m’) || (c >= ’A’ && c <= ’M’))

**return** c + 13;

**else if** ((c >= ’n’ && c <= ’z’) || (c >= ’N’ && c <= ’Z’)) **return** c - 13;

**else**

**return** c;

}

**void**

child(**int** fd)

{

**char** outbuf[MAX\_LINE+1];

size\_t outbuf\_used = 0;

ssize\_t result;

**while** (1) {

**char** ch;

result = recv(fd, &ch, 1, 0);

**if** (result == 0) {

**break**;

} **else if** (result == -1) { perror("read"); **break**;

}

/\* We do this test to keep the user from overflowing the buffer. \*/ **if** (outbuf\_used < **sizeof**(outbuf)) {

outbuf[outbuf\_used++] = rot13\_char(ch);

}

**if** (ch == ’\n’) {

send(fd, outbuf, outbuf\_used, 0);

outbuf\_used = 0;

**continue**;

}

}

}

**void**

run(**void**)

{

**int** listener;

**struct** sockaddr\_in sin;

sin.sin\_family = AF\_INET;

sin.sin\_addr.s\_addr = 0;

sin.sin\_port = htons(40713);

listener = socket(AF\_INET, SOCK\_STREAM, 0);

#ifndef WIN32

{

**int** one = 1;

setsockopt(listener, SOL\_SOCKET, SO\_REUSEADDR, &one, **sizeof**(one));

}

#endif

**if** (bind(listener, (**struct** sockaddr\*)&sin, **sizeof**(sin)) < 0) {perror("bind");

**return**;

}

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 6 / 120 |

**if** (listen(listener, 16)<0) {

perror("listen");

**return**;

}

**while** (1) {

**struct** sockaddr\_storage ss;

socklen\_t slen = **sizeof**(ss);

**int** fd = accept(listener, (**struct** sockaddr\*)&ss, &slen); **if** (fd < 0) {

perror("accept");

} **else** {

**if** (fork() == 0) {

child(fd);

exit(0);

}

}

}

}

**int**

main(**int** c, **char** \*\*v)

{

run();

**return** 0;

}

到现在我们是否已经找到了完美的处理多条连接的解决方案? 我是否能停止写本书去干点别的了？还不行。 创建进程或线程在很多操作系统中是一个非常大的消耗. 在现实中你可能会用线程池，但是线程是不能无限创建的, 如果你的程序需要同时处理上成千上万的连接。只拥有几个线程的CPU处理数万个线程时，会明显力不从心。

但是如果多线程不是解决多连接的方法，那么该怎么做？在Unix范例中，将你的socket设置为非阻塞的。系统调用是：

fcntl(fd, F\_SETFL, O\_NONBLOCK);

fd是你的套接字. [1](#page12)一旦你将socket设置为非阻塞,从此以后, 你的socket调用都会马上返回（有时候是调用完毕，有时候是出现了错误）例如返回错误 "I couldn’t make any progress now, try again." 所以我们的两个套接字的例子应该改写成这样：

Bad Example: busy-polling all sockets

/\* 这将有效, 但是性能会不可原谅地差 \*/

**int** i, n;

**char** buf[1024];

**for** (i=0; i < n\_sockets; ++i)

fcntl(fd[i], F\_SETFL, O\_NONBLOCK);

**while** (i\_still\_want\_to\_read()) {

**for** (i=0; i < n\_sockets; ++i) {

n = recv(fd[i], buf, **sizeof**(buf), 0);

**if** (n == 0) {

handle\_close(fd[i]);

} **else if** (n < 0) {

**if** (errno == EAGAIN)

* A file descriptor is the number the kernel assigns to the socket when you open it. You use this number to make Unix calls referring to the socket.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 7 / 120 |

; /\* The kernel didn’t have any data **for** us to read. \*/

**else**

handle\_error(fd[i], errno);

} **else** {

handle\_input(fd[i], buf, n);

}

}

}

现在我们使用了非阻塞的套接字, 上面的代码可以正常工作. . . 但是只是仅仅。性能会差到极致。有两个原因， 首先, 当两条连接都没有数据的时候，将不断地调用recv(), 这样会消耗完CPU的时钟周期。其次，如果你需要处理更多的套接字，每个周期内，不管有没有数据你每次都需要进行多次内核调用。 这样我们需要告诉内核 “阻塞直到有数据”并且告诉我们是哪些套接字有数据。

最古老的并且现在还在被使用的解决方案是select(). select() 调用有三个句柄集合 (使用bit数组实现): 一个是读、一个是写、一个是异常。它将阻塞程序，直到有一个集合准备好，并且只包含有读、写、异常的套接字。下面是我们使用select的示例：

Example: Using select

/\* If you only have a couple dozen fds, this version won’t be awful \*/ fd\_set readset;

**int** i, n;

**char** buf[1024];

**while** (i\_still\_want\_to\_read()) {

**int** maxfd = -1;

FD\_ZERO(&readset);

/\* Add all of the interesting fds to readset \*/

**for** (i=0; i < n\_sockets; ++i) {

**if** (fd[i]>maxfd) maxfd = fd[i];

FD\_SET(fd[i], &readset);

}

/\* Wait until one or more fds are ready to read \*/ select(maxfd+1, &readset, NULL, NULL, NULL);

/\* Process all of the fds that are still set in readset \*/ **for** (i=0; i < n\_sockets; ++i) {

**if** (FD\_ISSET(fd[i], &readset)) {

n = recv(fd[i], buf, **sizeof**(buf), 0);

**if** (n == 0) {

handle\_close(fd[i]);

} **else if** (n < 0) {

**if** (errno == EAGAIN)

; /\* The kernel didn’t have any data **for** us to read. \*/

**else**

handle\_error(fd[i], errno);

} **else** {

handle\_input(fd[i], buf, n);

}

}

}

}

下面是用select重新实现的 ROT13 服务器

Example: select()-based ROT13 server

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 8 / 120 |

/\* For sockaddr\_in \*/

#include <netinet/in.h>

/\* For socket functions \*/

#include <sys/socket.h>

/\* For fcntl \*/

#include <fcntl.h>

/\* for select \*/

#include <sys/select.h>

#include <assert.h>

#include <unistd.h>

#include <string.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#define MAX\_LINE 16384

**char**

rot13\_char(**char** c)

{

/\* We don’t want to use isalpha here; setting the locale would change

* which characters are considered alphabetical. \*/ **if** ((c >= ’a’ && c <= ’m’) || (c >= ’A’ && c <= ’M’))

**return** c + 13;

**else if** ((c >= ’n’ && c <= ’z’) || (c >= ’N’ && c <= ’Z’)) **return** c - 13;

**else**

**return** c;

}

**struct** fd\_state {

**char** buffer[MAX\_LINE];

size\_t buffer\_used;

**int** writing;

size\_t n\_written;

size\_t write\_upto;

};

**struct** fd\_state\*

alloc\_fd\_state(**void**)

{

**struct** fd\_state\*state = malloc(**sizeof**(**struct** fd\_state)); **if** (!state)

**return** NULL;

state->buffer\_used = state->n\_written = state->writing = state->write\_upto = 0;

**return** state;

}

**void**

free\_fd\_state(**struct** fd\_state \*state)

{

free(state);

}

**void**

make\_nonblocking(**int** fd)

{

fcntl(fd, F\_SETFL, O\_NONBLOCK);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 9 / 120 |

}

**int**

do\_read(**int** fd, **struct** fd\_state \*state)

{

**char** buf[1024];

**int** i;

ssize\_t result;

**while** (1) {

result = recv(fd, buf, **sizeof**(buf), 0);

**if** (result <= 0)

**break**;

**for** (i=0; i < result; ++i) {

**if** (state->buffer\_used < **sizeof**(state->buffer))state->buffer[state->buffer\_used++] = rot13\_char(buf[i]);

**if** (buf[i] == ’\n’) {

state->writing = 1;

state->write\_upto = state->buffer\_used;

}

}

}

**if** (result == 0) {

**return** 1;

} **else if** (result < 0) {

**if** (errno == EAGAIN)

**return** 0;

**return** -1;

}

**return** 0;

}

**int**

do\_write(**int** fd, **struct** fd\_state \*state)

{

**while** (state->n\_written < state->write\_upto) {

ssize\_t result = send(fd, state->buffer + state->n\_written, state->write\_upto - state->n\_written, 0);

**if** (result < 0) {

**if** (errno == EAGAIN)

**return** 0;

**return** -1;

}

assert(result != 0);

state->n\_written += result;

}

**if** (state->n\_written == state->buffer\_used)

state->n\_written = state->write\_upto = state->buffer\_used = 0;

state->writing = 0;

**return** 0;

}

**void**

run(**void**)

{

**int** listener;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Fast portable non-blocking network programming with Libevent | | | 10 / 120 | |
|  |  |  |  |  |
|  | **struct** | fd\_state \*state[FD\_SETSIZE]; |  |  |
|  | **struct** | sockaddr\_in sin; |  |  |
|  | **int** i, | maxfd; |  |  |
|  | fd\_set | readset, writeset, exset; |  |  |

sin.sin\_family = AF\_INET;

sin.sin\_addr.s\_addr = 0;

sin.sin\_port = htons(40713);

**for** (i = 0; i < FD\_SETSIZE; ++i)

state[i] = NULL;

listener = socket(AF\_INET, SOCK\_STREAM, 0);

make\_nonblocking(listener);

#ifndef WIN32

{

**int** one = 1;

setsockopt(listener, SOL\_SOCKET, SO\_REUSEADDR, &one, **sizeof**(one));

}

#endif

**if** (bind(listener, (**struct** sockaddr\*)&sin, **sizeof**(sin)) < 0) {perror("bind");

**return**;

}

**if** (listen(listener, 16)<0) {

perror("listen");

**return**;

}

FD\_ZERO(&readset);

FD\_ZERO(&writeset);

FD\_ZERO(&exset);

**while** (1) {

maxfd = listener;

FD\_ZERO(&readset);

FD\_ZERO(&writeset);

FD\_ZERO(&exset);

FD\_SET(listener, &readset);

**for** (i=0; i < FD\_SETSIZE; ++i) {

**if** (state[i]) {

**if** (i > maxfd)

maxfd = i;

FD\_SET(i, &readset);

**if** (state[i]->writing) {

FD\_SET(i, &writeset);

}

}

}

**if** (select(maxfd+1, &readset, &writeset, &exset, NULL) < 0) {perror("select");

**return**;

}

**if** (FD\_ISSET(listener, &readset)) {

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 11 / 120 |

**struct** sockaddr\_storage ss;

socklen\_t slen = **sizeof**(ss);

**int** fd = accept(listener, (**struct** sockaddr\*)&ss, &slen); **if** (fd < 0) {

perror("accept");

} **else if** (fd > FD\_SETSIZE) { close(fd);

} **else** {

make\_nonblocking(fd);

state[fd] = alloc\_fd\_state();

assert(state[fd]);/\*XXX\*/

}

}

**for** (i=0; i < maxfd+1; ++i) {

**int** r = 0;

**if** (i == listener)

**continue**;

**if** (FD\_ISSET(i, &readset)) {

r = do\_read(i, state[i]);

}

**if** (r == 0 && FD\_ISSET(i, &writeset)) {

r = do\_write(i, state[i]);

}

**if** (r) {

free\_fd\_state(state[i]);

state[i] = NULL;

close(i);

}

}

}

}

**int**

main(**int** c, **char** \*\*v)

{

setvbuf(stdout, NULL, \_IONBF, 0);

run();

**return** 0;

}

但是我们还没有完工. 因为将套接字加入到fd集合中的时间是和套接字数据成比例增加的。当连接非常多时，性能下降严重。 [2](#page17)

不同的操作系统提供了不同的调用来替代select。包括poll(), epoll(), kqueue(), evports, 和 /dev/poll。所有这些调用都比select性能好。特别是poll() 提供了O(1) 的时间复杂度来处理套接字的读写。

不幸的是，每种API都不是标准的。Linux 有 epoll(), BSDs (Darwin) have kqueue(), Solaris 有evports 和/dev/poll. . . 都不能通用。所以如果你 想写高性能的跨平台异步调用程序，你需要一个将不同操作系统上调用包装起来的抽象。这正是libevent最底层API为你做的。它提供了一个一致的接口去包装每个操作系统上性能最高的接口。

下面是另一个版本的 ROT13 服务器. 这回它使用Libevent 2 替换select(). 记住

* On the userspace side, generating and reading the bit arrays can be made to take time proportional to the number of fds that you provided for select(). But on the kernel side, reading the bit arrays takes time proportional to the largest fd in the bit array, which tends to be around the total number of fds in use in the whole program, regardless of how many fds are added to the sets in select().

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 12 / 120 |

fd\_sets 已经不见了: 取而代之的是, 我们关联、撤销关联事件和event\_base, 它将会去调用 select(), poll(), epoll(), kqueue()等等。

Example: A low-level ROT13 server with Libevent

/\* For sockaddr\_in \*/

#include <netinet/in.h>

/\* For socket functions \*/

#include <sys/socket.h>

/\* For fcntl \*/

#include <fcntl.h>

#include <event2/event.h>

#include <assert.h>

#include <unistd.h>

#include <string.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#define MAX\_LINE 16384

**void** do\_read(evutil\_socket\_t fd, **short** events, **void** \*arg); **void** do\_write(evutil\_socket\_t fd, **short** events, **void** \*arg);

**char**

rot13\_char(**char** c)

{

/\* We don’t want to use isalpha here; setting the locale would change

* which characters are considered alphabetical. \*/ **if** ((c >= ’a’ && c <= ’m’) || (c >= ’A’ && c <= ’M’))

**return** c + 13;

**else if** ((c >= ’n’ && c <= ’z’) || (c >= ’N’ && c <= ’Z’)) **return** c - 13;

**else**

**return** c;

}

**struct** fd\_state {

**char** buffer[MAX\_LINE];

size\_t buffer\_used;

size\_t n\_written;

size\_t write\_upto;

**struct** event\*read\_event;

**struct** event\*write\_event;

};

**struct** fd\_state\*

alloc\_fd\_state(**struct** event\_base \*base, evutil\_socket\_t fd)

{

**struct** fd\_state\*state = malloc(**sizeof**(**struct** fd\_state)); **if** (!state)

**return** NULL;

state->read\_event = event\_new(base, fd, EV\_READ|EV\_PERSIST, do\_read, state);

**if** (!state->read\_event) {

free(state);

**return** NULL;

}

state->write\_event =

event\_new(base, fd, EV\_WRITE|EV\_PERSIST, do\_write, state);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 13 / 120 |

**if** (!state->write\_event) {

event\_free(state->read\_event);

free(state);

**return** NULL;

}

state->buffer\_used = state->n\_written = state->write\_upto = 0;

assert(state->write\_event);

**return** state;

}

**void**

free\_fd\_state(**struct** fd\_state \*state)

{

event\_free(state->read\_event);

event\_free(state->write\_event);

free(state);

}

**void**

do\_read(evutil\_socket\_t fd, **short** events, **void** \*arg)

{

**struct** fd\_state\*state = arg;

**char** buf[1024];

**int** i;

ssize\_t result;

**while** (1) {

assert(state->write\_event);

result = recv(fd, buf, **sizeof**(buf), 0);

**if** (result <= 0)

**break**;

**for** (i=0; i < result; ++i) {

**if** (state->buffer\_used < **sizeof**(state->buffer))state->buffer[state->buffer\_used++] = rot13\_char(buf[i]);

**if** (buf[i] == ’\n’) {

assert(state->write\_event);

event\_add(state->write\_event, NULL);

state->write\_upto = state->buffer\_used;

}

}

}

**if** (result == 0) {

free\_fd\_state(state);

} **else if** (result < 0) {

**if** (errno == EAGAIN)// XXXX use evutil macro

**return**;

perror("recv");

free\_fd\_state(state);

}

}

**void**

do\_write(evutil\_socket\_t fd, **short** events, **void** \*arg)

{

**struct** fd\_state\*state = arg;

**while** (state->n\_written < state->write\_upto) {

ssize\_t result = send(fd, state->buffer + state->n\_written,

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 14 / 120 |

state->write\_upto - state->n\_written, 0);

**if** (result < 0) {

**if** (errno == EAGAIN)// XXX use evutil macro

**return**;

free\_fd\_state(state);

**return**;

}

assert(result != 0);

state->n\_written += result;

}

**if** (state->n\_written == state->buffer\_used)

state->n\_written = state->write\_upto = state->buffer\_used = 1;

event\_del(state->write\_event);

}

**void**

do\_accept(evutil\_socket\_t listener, **short** event, **void** \*arg)

{

**struct** event\_base\*base = arg;

**struct** sockaddr\_storage ss;

socklen\_t slen = **sizeof**(ss);

**int** fd = accept(listener, (**struct** sockaddr\*)&ss, &slen); **if** (fd < 0) {// XXXX eagain??

perror("accept");

} **else if** (fd > FD\_SETSIZE) {

close(fd); // XXX replace all closes with EVUTIL\_CLOSESOCKET \*/

} **else** {

**struct** fd\_state\*state;

evutil\_make\_socket\_nonblocking(fd);

state = alloc\_fd\_state(base, fd);

assert(state); /\*XXX err\*/

assert(state->write\_event);

event\_add(state->read\_event, NULL);

}

}

**void**

run(**void**)

{

evutil\_socket\_t listener;

**struct** sockaddr\_in sin;

**struct** event\_base\*base;

**struct** event\*listener\_event;

base = event\_base\_new();

**if** (!base)

**return**; /\*XXXerr\*/

sin.sin\_family = AF\_INET;

sin.sin\_addr.s\_addr = 0;

sin.sin\_port = htons(40713);

listener = socket(AF\_INET, SOCK\_STREAM, 0);

evutil\_make\_socket\_nonblocking(listener);

#ifndef WIN32

{

**int** one = 1;

setsockopt(listener, SOL\_SOCKET, SO\_REUSEADDR, &one, **sizeof**(one));

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 15 / 120 |

}

#endif

**if** (bind(listener, (**struct** sockaddr\*)&sin, **sizeof**(sin)) < 0) {perror("bind");

**return**;

}

**if** (listen(listener, 16)<0) {

perror("listen");

**return**;

}

listener\_event = event\_new(base, listener, EV\_READ|EV\_PERSIST, do\_accept, (**void**\*)base); /\*XXX check it \*/

event\_add(listener\_event, NULL);

event\_base\_dispatch(base);

}

**int**

main(**int** c, **char** \*\*v)

{

setvbuf(stdout, NULL, \_IONBF, 0);

run();

**return** 0;

}

(我们使用evutil\_socket\_t替换了int来表示socket. 调用evutil\_make\_socket\_nonblocking替换fcntl(O\_NONBLOCK) 使套接字非阻塞。这些变化使得我们的代码可以在Windows上执行)

**易用性怎么样? (Windows?)**

或许你已经注意到，我们的代码已经变得更加有效率，它也变得更加复杂了。当我们之前调用fork的时候，我们不需要管理每个连接的buffer。我们只是有一个在栈上分配的buffer。我们不需要显式地去跟踪每个套接字是否可写，是否可读。我们也不需要一个结构去跟踪有多少操作已经完成，我们仅仅使用了栈上的变量。

如果你熟悉Windows网络编程，你将会发现Libevent 在Windows并没有提升性能。在Windows上最快的异步IO方式不是像select()一样的调用。而是使用完成端口(IOCP)。与其它API都不同的是，完成端口不会在套接字可操作的时候通知你的程序，而是会让协议栈去处理读、写等操作，操作完成时才会通知到应用程序。

幸运的是，Libevent 2的 "bufferevents"接口解决了这些问题，它使得应用程序编写起来更加简单。它提供了一个在Windows上和Unix上都能高效地运行的接口。

下面是最终版的使用了bufferevents API的 ROT13 服务器。

Example: A simpler ROT13 server with Libevent

/\* For sockaddr\_in \*/

#include <netinet/in.h>

/\* For socket functions \*/

#include <sys/socket.h>

/\* For fcntl \*/

#include <fcntl.h>

#include <event2/event.h>

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 16 / 120 |

#include <event2/buffer.h>

#include <event2/bufferevent.h>

#include <assert.h>

#include <unistd.h>

#include <string.h>

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#define MAX\_LINE 16384

**void** do\_read(evutil\_socket\_t fd, **short** events, **void** \*arg); **void** do\_write(evutil\_socket\_t fd, **short** events, **void** \*arg);

**char**

rot13\_char(**char** c)

{

/\* We don’t want to use isalpha here; setting the locale would change

* which characters are considered alphabetical. \*/ **if** ((c >= ’a’ && c <= ’m’) || (c >= ’A’ && c <= ’M’))

**return** c + 13;

**else if** ((c >= ’n’ && c <= ’z’) || (c >= ’N’ && c <= ’Z’)) **return** c - 13;

**else**

**return** c;

}

**void**

readcb(**struct** bufferevent \*bev, **void** \*ctx)

{

**struct** evbuffer\*input,\*output;

**char** \*line;

size\_t n;

**int** i;

input = bufferevent\_get\_input(bev);

output = bufferevent\_get\_output(bev);

**while** ((line = evbuffer\_readln(input, &n, EVBUFFER\_EOL\_LF))) { **for** (i = 0; i < n; ++i)

line[i] = rot13\_char(line[i]);

evbuffer\_add(output, line, n);

evbuffer\_add(output, "\n", 1);

free(line);

}

**if** (evbuffer\_get\_length(input) >= MAX\_LINE) {

/\* Too long; just process what there is and go on so that the buffer

* doesn’t grow infinitely long. \*/ **char** buf[1024];

**while** (evbuffer\_get\_length(input)) {

**int** n = evbuffer\_remove(input, buf, **sizeof**(buf)); **for** (i = 0; i < n; ++i)

buf[i] = rot13\_char(buf[i]); evbuffer\_add(output, buf, n);

}

evbuffer\_add(output, "\n", 1);

}

}

**void**

errorcb(**struct** bufferevent \*bev, **short** error, **void** \*ctx)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Fast portable non-blocking network programming with Libevent | | 17 / 120 | |
|  |  |  |  |  |
|  | { |  |  |  |
|  | **if** (error & BEV\_EVENT\_EOF) { | |  |  |
|  | /\* | connection has been closed, do any clean up here \*/ |  |  |
|  | /\* | ... \*/ |  |  |
|  | } **else** | **if** (error & BEV\_EVENT\_ERROR) { |  |  |
|  | /\* | check errno to see what error occurred \*/ |  |  |
|  | /\* | ... \*/ |  |  |
|  | } **else** | **if** (error & BEV\_EVENT\_TIMEOUT) { |  |  |
|  | /\* | must be a timeout event handle, handle it \*/ |  |  |
|  | /\* | ... \*/ |  |  |
|  | } |  |  |  |

bufferevent\_free(bev);

}

**void**

do\_accept(evutil\_socket\_t listener, **short** event, **void** \*arg)

{

**struct** event\_base\*base = arg;

**struct** sockaddr\_storage ss;

socklen\_t slen = **sizeof**(ss);

**int** fd = accept(listener, (**struct** sockaddr\*)&ss, &slen); **if** (fd < 0) {

perror("accept");

} **else if** (fd > FD\_SETSIZE) { close(fd);

} **else** {

**struct** bufferevent\*bev;

evutil\_make\_socket\_nonblocking(fd);

bev = bufferevent\_socket\_new(base, fd, BEV\_OPT\_CLOSE\_ON\_FREE); bufferevent\_setcb(bev, readcb, NULL, errorcb, NULL); bufferevent\_setwatermark(bev, EV\_READ, 0, MAX\_LINE); bufferevent\_enable(bev, EV\_READ|EV\_WRITE);

}

}

**void**

run(**void**)

{

evutil\_socket\_t listener;

**struct** sockaddr\_in sin;

**struct** event\_base\*base;

**struct** event\*listener\_event;

base = event\_base\_new();

**if** (!base)

**return**; /\*XXXerr\*/

sin.sin\_family = AF\_INET;

sin.sin\_addr.s\_addr = 0;

sin.sin\_port = htons(40713);

listener = socket(AF\_INET, SOCK\_STREAM, 0);

evutil\_make\_socket\_nonblocking(listener);

#ifndef WIN32

{

**int** one = 1;

setsockopt(listener, SOL\_SOCKET, SO\_REUSEADDR, &one, **sizeof**(one));

}

#endif

**if** (bind(listener, (**struct** sockaddr\*)&sin, **sizeof**(sin)) < 0) {

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 18 / 120 |

perror("bind");

**return**;

}

**if** (listen(listener, 16)<0) {

perror("listen");

**return**;

}

listener\_event = event\_new(base, listener, EV\_READ|EV\_PERSIST, do\_accept, (**void**\*)base); /\*XXX check it \*/

event\_add(listener\_event, NULL);

event\_base\_dispatch(base);

}

**int**

main(**int** c, **char** \*\*v)

{

setvbuf(stdout, NULL, \_IONBF, 0);

run();

**return** 0;

}

**这些代码效率怎么样?**

XXXX 写下了这段高效率的代码。 libevent官网上的性能测试数据已经非常老了。

**Libevent 参考手册: 前言**

**Libevent from 10,000 feet**

Libevent 是一个编写高性能跨平台非阻塞网络程序的库。它的设计目标如下：

跨平台

使用Libevent写的应用程序可以在所有支持Libevent的操作系统上运行。即使没有好的办法去使用非阻塞IO，Libevent也会支持得差不多。所以你的程序能够跑在 受限制的操作系统上。

性能

Libevent尝试使用每种操作系统上最快的非阻塞调用。并不打算使用新的做法来替换当前做法。

可扩展性

Libevent 为了万级的SOCKET而设计。

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 19 / 120 |

易用性

不管怎么样，正常使用 Libevent都必须保证稳定, 可移植。

Libevent被分为以下几个组件：

evutil

不同操作系统上通用函数的封装。

event and event\_base

这是Libevent的核心。它提供了屏蔽各种操作系统差异的非阻塞IO抽象API 。它能在socket可读写的时候够通知你、基本的超时函数、检测操作系统信号。

bufferevent

这些函数提供了更方便的Libevent’s event-based 核心包装接口。他们让你的程序能够自动地读写socket，而不是通知你。当IO真实实现的时候才会通知你。

The bufferevent interface also has multiple backends, so that it can take advantage of systems that provide faster ways to do nonblocking IO, such as the Windows IOCP API.

evbuffer

这个模块实现 bufferevents底下的buffers，并且提供高效而方便的访问接口。

evhttp

一个简单的 HTTP client/server 实现。

evdns

一个简单的DNS client/server 实现。

evrpc

一个简单的RPC 实现。

**The Libraries**

编译Libevent的时候，它会生成以下这些库：

libevent\_core

所有核心事件 buffer 函数. 这个库包含所有的event\_base, evbuffer, bufferevent, 和工具函数。

libevent\_extra

它包含了一些网络协议，如 HTTP, DNS, and RPC。

libevent

这个库是为了兼容历史版本而存在。它包含libevent\_core 和libevent\_extra。你不应该使用它。将来libevent会删除它。

下面的这些库只会在部分操作系统中生成：

libevent\_pthreads

这个库加入了依赖pthreads库的线程和锁实现。它与libevent\_core是相互独立 的，所以你不需要链接pthread库，除非你想采用多线程的方式使用libevent。

libevent\_openssl

这个库使用bufferevents 和 OpenSSL来支持加密通信。它与 libevent\_core是相互独立的，所以你不需要链接 OpenSSL，除非你想使用加密通信。

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 20 / 120 |

**头文件**

所有当前 Libevent 头文件安装在event2 目录。有三种类型的头文件：

API headers

一个API头文件包含libevent公开接口的定义。这些文件没有特殊的后缀。

Compatibility headers

兼容性头文件包含老版本的函数定义. 除非你使用的是老版本libevent，否则你不用包含它们。

Structure headers

这些头文件包含相关结构的定义。有些是公开的，为了方便你查看。有些是因为历史原因。依赖这些结构会让你的程序变得不方便移植，变得不可调试。这些文件的后缀是"\_struct.h"

(There are also older versions of the Libevent headers without the event2 directory. See "If you have to work with an old version of Libevent" below.)

**如果你必须兼容低版本的libevent**

Libevent 2.0 将APIs 修正得更合理，更容易处理错误。你需要使用 Libevent 2.0 APIs。有时你可能需要兼容老的版本。

老版本的libevent拥有很少的头文件, 没有安装在 "event2"目录下：

|  |  |
| --- | --- |
| OLD HEADER. . . | . . . REPLACED BY CURRENT HEADERS |
| event.h | event2/event\*.h, event2/buffer\*.h event2/bufferevent\*.h |
|  | event2/tag\*.h |
| evdns.h | event2/dns\*.h |
| evhttp.h | event2/http\*.h |
| evrpc.h | event2/rpc\*.h |
| evutil.h | event2/util\*.h |

在Libevent 2.0以后的版本中，老版本头文件实际上是对新头文件的包装。

一些其它的老版本的兼容性建议:

* 在1.4版本前只有一个库： "libevent"。它包含的函数现在被分在了 libevent\_core 和 libevent\_extra。
* 2.0版本前，不支持锁。只有在两个线程不同时访问同一结构时才能保证线程案例。

下面我们将会讨论独立的API，这些API在特定的代码段中。

**版本状态**

在1.4.7版本前是完全独立的。1.3e版本 拥有很多bug。

(Also, please don’t send the Libevent maintainers any new features for 1.4.x or earlier---it’s supposed to stay as a stable release. And if you encounter a bug in 1.3x or earlier, please make sure that it still exists in the latest stable version before you report it: subsequent releases have happened for a reason.)

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 21 / 120 |

**安装Libevent库**

Libevent 有一些全局共享的变量。它们影响整个库。

你必须在调用API前修改这些变量，否则Libevent将会变得行为诡异。

**Libevent日志记录**

Libevent 可以记录内部错误和警告。在编译了logging后，能够记录调试信息。默认情况下这些信息会写到错误终端。你可以通过提供自己的日志函数来替换这一行为。

接口

#define EVENT\_LOG\_DEBUG 0

#define EVENT\_LOG\_MSG 1

#define EVENT\_LOG\_WARN 2

#define EVENT\_LOG\_ERR 3

/\* Deprecated; see note at the end of this section \*/ #define \_EVENT\_LOG\_DEBUG EVENT\_LOG\_DEBUG

#define \_EVENT\_LOG\_MSG EVENT\_LOG\_MSG

#define \_EVENT\_LOG\_WARN EVENT\_LOG\_WARN

#define \_EVENT\_LOG\_ERR EVENT\_LOG\_ERR

**typedef void** (\*event\_log\_cb)(**int** severity, **const char** \*msg);

**void** event\_set\_log\_callback(event\_log\_cb cb);

你可以写一个回调函数，并将之作为参数调用event\_set\_log\_callback().当Libevent 写日志时，它会调用你提供的回调函数。你可以调用event\_set\_log\_callback(NULL)恢复成默认处理。

Examples

#include <event2/event.h>

#include <stdio.h>

**static void** discard\_cb(**int** severity, **const char** \*msg)

{

/\* This callback does nothing. \*/

}

**static** FILE\*logfile = NULL;

**static void** write\_to\_file\_cb(**int** severity, **const char** \*msg)

{

**const char** \*s;

**if** (!logfile)

**return**;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Fast portable non-blocking network programming with Libevent | | | | 22 / 120 | |
|  |  |  |  |  |  |
|  | **switch** (severity) { |  |  |  |  |
|  | **case** \_EVENT\_LOG\_DEBUG: s = "debug"; | | **break**; |  |  |
|  | **case** \_EVENT\_LOG\_MSG: | s = "msg"; | **break**; |  |  |
|  | **case** \_EVENT\_LOG\_WARN: | s = "warn"; | **break**; |  |  |
|  | **case** \_EVENT\_LOG\_ERR: | s = "error"; | **break**; |  |  |
|  | **default**: | s = "?"; | **break**; /\*never reached\*/ |  |  |
| } | |  |  |  |  |

fprintf(logfile, "[%s] %s\n", s, msg);

}

/\* Turn off all logging from Libevent. \*/

**void** suppress\_logging(**void**)

{

event\_set\_log\_callback(discard\_cb);

}

/\* Redirect all Libevent log messages to the C stdio file ’f’. \*/

**void** set\_logfile(FILE\*f)

{

logfile = f;

event\_set\_log\_callback(write\_to\_file\_cb);

}

记住在回调函数中不要调用libevent的函数。例如：你的回调函数使用bufferevents将错误信息发送出去，你的程序可能会表现得很诡异，并且很难诊断中错误。在部分函数中，这个约束将在后面的版本删掉。

按说，调试日志是非激活的，不会发送到日志回调中。如果libevent在编译的时候包含了，你可以手动打开它。

Interface

#define EVENT\_DBG\_NONE 0

#define EVENT\_DBG\_ALL 0xffffffffu

**void** event\_enable\_debug\_logging(ev\_uint32\_t which);

调试日志很多，在很多情况下都是不必要的。调用event\_enable\_debug\_logging(EVENT\_DBG\_NONE) 会恢复默认行为。使用EVENT\_DBG\_ALL将会打开所有支持的日志。更细致的选项支持会在将来的版本中提供。

这些函数定义在<event2/event.h>。它们第一次是出现在Libevent 1.0c，event\_enable\_debug\_logging()第一次出现在Libevent 2.1.1-alpha。

兼容性提示 Libevent 2.0.19-stable之前，EVENT\_LOG\_\*开头有下划线: \_EVENT\_LOG\_DEBUG, \_EVENT\_LOG\_MSG、\_EVENT\_LOG\_WARN、\_EVENT\_LOG\_ERR。 这些老名称是不建议使用的。它们将会在将来的版本中被删除。

**处理严重错误**

当Libevent检测到一个没能恢复的内部错误（比如一个不正确的数据结构）它默认的处理方式是调用exit()或者abort()去结束进程. 这些错误通常意味着某处代码有BUG。可能是你的代码，也可能是Libevent的代码。

你可以通过提供自己的错误处理函数来优雅地处理错误，Libevent会在错误发生时调用它。

Interface

**typedef void** (\*event\_fatal\_cb)(**int** err);

**void** event\_set\_fatal\_callback(event\_fatal\_cb cb);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 23 / 120 |

首先你需要定义错误处理函数。然后使用它作为参数调用 event\_set\_fatal\_callback()。如果Libevent发生了一个严重错误，它将出调用你提供的函数。

你的函数不应该返回到Libevent的控制流程，这样做会导致不可知的行为。Libevent很可能直接退出避免直接crash。一旦你的函数被调用，你不应该再调用任何Libevent的函数。

这些函数定义在<event2/event.h>。它们第一次出现是在Libevent 2.0.3-alpha。

**内存管理**

默认情况下，Libevent使用C库的内存管理函数来动态分配内存。你可以使用自己的分配函数。你可能有一个更高效率的分配器，并且你希望 Libevent使用它。或者你的内存分配器能够检测内存泄露。

Interface

|  |  |
| --- | --- |
| **void** event\_set\_mem\_functions(**void** | \*(\*malloc\_fn)(size\_t sz), |
| **void** | \*(\*realloc\_fn)(**void** \*ptr, size\_t sz), |
| **void** | (\*free\_fn)(**void** \*ptr)); |
|  |  |

这是一个简单的例子，Libevent使用了新的分配器，这个分配器能够统计分配的内存总量。如果程序运行在多线程环境中，你需要加锁。

Example

#include <event2/event.h>

#include <sys/types.h>

#include <stdlib.h>

/\* This union’s purpose is to be as big as the largest of all the

* types it contains. \*/ **union** alignment {

size\_t sz; **void** \*ptr; **double** dbl;

};

/\* We need to make sure that everything we return is on the right alignment to hold anything, including a double. \*/

#define ALIGNMENT **sizeof**(**union** alignment)

/\* We need to do this cast-to-char\* trick on our pointers to adjust them; doing arithmetic on a void\* is not standard. \*/

#define OUTPTR(ptr) (((**char**\*)ptr)+ALIGNMENT)

#define INPTR(ptr) (((**char**\*)ptr)-ALIGNMENT)

**static** size\_t total\_allocated = 0;

**static void** \*replacement\_malloc(size\_t sz)

{

**void** \*chunk = malloc(sz + ALIGNMENT);

**if** (!chunk) **return** chunk;

total\_allocated += sz;

\*(size\_t\*)chunk = sz;

**return** OUTPTR(chunk);

}

**static void** \*replacement\_realloc(**void** \*ptr, size\_t sz)

{

size\_t old\_size = 0;

**if** (ptr) {

ptr = INPTR(ptr);

old\_size = \*(size\_t\*)ptr;

}

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 24 / 120 |

ptr = realloc(ptr, sz + ALIGNMENT);

**if** (!ptr)

**return** NULL;

\*(size\_t\*)ptr = sz;

total\_allocated = total\_allocated - old\_size + sz; **return** OUTPTR(ptr);

}

**static void** replacement\_free(**void** \*ptr)

{

ptr = INPTR(ptr);

total\_allocated -= \*(size\_t\*)ptr;

free(ptr);

}

**void** start\_counting\_bytes(**void**)

{

event\_set\_mem\_functions(replacement\_malloc,

replacement\_realloc,

replacement\_free);

}

**提示**

* 替换函数会影响Libevent所有的内存分配。你必须在调用Libevent任何其它函数之前来替换。否则Libevent会使用你的free函数去释放之前用C库分配的内存。
* 你的malloc 和realloc 返回的内存，内存对齐情况要和C库的一样。
* 你的重分配函数必须正确处理 realloc(NULL, sz) (当作malloc(sz))。
* 你的realloc 必须正确处理realloc(ptr, 0) (当作 free(ptr))。
* 你的函数不处理 free(NULL)。
* 你的函数不处理malloc(0)。
* 在多线程环境下，你提供的函数必须注意线程安全。
* Libevent分配的内存会返回给你的程序。你必须使用Libevent的函数去释放它，或者使用相同的释放函数。

event\_set\_mem\_functions() 定义在 <event2/event.h>。它第一次出现是在 Libevent 2.0.1-alpha。

Libevent在编译的时候可以禁止event\_set\_mem\_functions() 。这样event\_set\_mem\_functions将不会链接进程序。Libevent 2.0.2-alpha版本后，你可以通过检测EVENT\_SET\_MEM\_FUNCTIONS\_IMPLEMENTED 宏来检查该函数是否可用。

**线程和锁**

如你所知，多个线程同时访问相同的对象，经常是不安全的。

Libevent 的结构可以与多线程有三种情况：

* 有些结构永远只能用在单一线程中。
* 有些对象访问的时候可以加锁:你必须显示地告诉Libevent哪些对象需要加锁。
* 有些对象访问的时候会加上锁，永远是线程安全的。

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 25 / 120 |

你必须设置加锁函数，告诉 Libevent 使用哪些加锁函数。你必须在调用任何需要加锁的函数前设置。如果你使用的是pthread库或者Windows的api，你是幸运的。这些预定义的函数可以帮助你进行设置。

Interface

#ifdef WIN32

**int** evthread\_use\_windows\_threads(**void**);

#define EVTHREAD\_USE\_WINDOWS\_THREADS\_IMPLEMENTED #endif

#ifdef \_EVENT\_HAVE\_PTHREADS

**int** evthread\_use\_pthreads(**void**);

#define EVTHREAD\_USE\_PTHREADS\_IMPLEMENTED

#endif

这些函数返回 0表示成功， -1表示失败。

如果你想使用不同的线程库，你必须为你使用的库写包装函数：

* Locks
* locking
* unlocking
* lock allocation
* lock destruction
* Conditions
* condition variable creation
* condition variable destruction
* waiting on a condition variable
* signaling/broadcasting to a condition variable
* Threads
* thread ID detection

然后你 必须使用evthread\_set\_lock\_callbacks 和evthread\_set\_id\_callback将这些函数注册到Libevent 。

Interface

|  |  |  |  |
| --- | --- | --- | --- |
| #define EVTHREAD\_WRITE | | | 0x04 |
|  | #define | EVTHREAD\_READ | 0x08 |
| #define | | EVTHREAD\_TRY | 0x10 |

#define EVTHREAD\_LOCKTYPE\_RECURSIVE 1

#define EVTHREAD\_LOCKTYPE\_READWRITE 2

#define EVTHREAD\_LOCK\_API\_VERSION 1

**struct** evthread\_lock\_callbacks {

**int** lock\_api\_version;

**unsigned** supported\_locktypes;

**void** \*(\*alloc)(**unsigned** locktype);

**void** (\*free)(**void** \*lock, **unsigned** locktype);

**int** (\*lock)(**unsigned** mode, **void** \*lock);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 26 / 120 |

**int** (\*unlock)(**unsigned** mode, **void** \*lock);

};

**int** evthread\_set\_lock\_callbacks(**const struct** evthread\_lock\_callbacks\*);

**void** evthread\_set\_id\_callback(**unsigned long** (\*id\_fn)(**void**));

**struct** evthread\_condition\_callbacks {

**int** condition\_api\_version;

**void** \*(\*alloc\_condition)(**unsigned** condtype);

**void** (\*free\_condition)(**void** \*cond);

**int** (\*signal\_condition)(**void** \*cond, **int** broadcast); **int** (\*wait\_condition)(**void** \*cond, **void** \*lock,

**const struct** timeval\*timeout);

};

**int** evthread\_set\_condition\_callbacks(

**const struct** evthread\_condition\_callbacks\*);

evthread\_lock\_callbacks 结构记录了你的线程函数。在以上版本中lock\_api\_version field必须被设置成EVTHREAD\_LOCK\_API\_VERSION。supported\_locktypes必须被设置成一个 EVTHREAD\_LOCKTYPE\_\* ，它们描述了你的锁的功能。 (在2.0.4-alpha, EVTHREAD\_LOCK\_RECURSIVE 是强制的， EVTHREAD\_LOCK\_READWRITE是无用的) alloc 函数必须返回一个指定的新锁对象。Free函数必须释放锁对象相关的所有资源。锁函数必须尝试获取特定的锁，成功时返回0，失败时返回非0。Unlock函数尝试unlock锁，成功返回0，失败返回非0。

锁类型如下:

0

A regular, not-necessarily recursive lock.

EVTHREAD\_LOCKTYPE\_RECURSIVE

线程可重入，同一线程多次锁不会再阻塞。 别的线程必须等锁解开（锁多少次就要解多少次）后才能获取到锁。

EVTHREAD\_LOCKTYPE\_READWRITE

读写锁

Recognized lock modes are:

EVTHREAD\_READ

只用在READWRITE: 请求读。

EVTHREAD\_WRITE

只用在READWRITE: 请求写。

EVTHREAD\_TRY

只用于锁: 尝试获取锁

id\_fn 是一个返回线程ID的函数。对于同样的线程，它必须返回同样的数字。不同的线程返回不同的数字。

evthread\_condition\_callbacks记录条件变量。在上面提到的版本中，lock\_api\_version 必须被设置成EVTHREAD\_CONDITION\_API\_VERSION。alloc\_condition必须返回指向新条件对象的指针。它接受0作为参数。free\_condition 必须释放条件变量相关的资源。wait\_condition接受三个参数: alloc\_condition分配的条件变量指针， evthread\_lock\_callbacks.alloc 生成的锁，一个超时时间。这个锁将会全程被持有。

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 27 / 120 |

函数必须释放锁，等条件变量触发或超时。wait\_condition在发生错误的时候返回-1，当条件变量触发返回0，超时时返回1 。返回时必须确认重新获取到了锁。最终signal\_condition 会唤醒等待条件变量的线程(如果broadcast参数是false) 或者是所有线程 (如果broadcast 参数是 true)。它只会在持有关联的锁的时候被持有。

关于条件变量的更多信息，你可以看pthreads’s pthread\_cond\_\* functions或者Windows的 CONDITION\_VARIABLE 函数。

Examples

如何使用这些的例子，参见Libevent的源代码中的evthread\_pthread.c 和evthread\_win32.c

本节中讨论的函数在<event2/thread.h>。大多数首次出现在Libevent 2.0.4-alpha。2.0.1-alpha 到2.0.3-alpha 使用老版的接口设置锁函数。event\_use\_pthreads()需要你的程序链接了event\_pthreads 库。

条件变量最新版在Libevent 2.0.7-rc。它们用来解决一些死锁问题。

Libevent 可以编译配置成无锁支持。这样，如果你的程序使用了上面的函数，将不能运行。

**调试锁用法**

为了能够调试锁，Libevent 有一个"lock debugging" 可选项。为了捕捉到锁错误，它包装了所有锁操作，包括:

* 解锁我们未曾获取的锁
* 重复锁非可重入锁

当这些错误发生时， Libevent 会打印断言并退出。

Interface

**void** evthread\_enable\_lock\_debugging(**void**);

#define evthread\_enable\_lock\_debuging() evthread\_enable\_lock\_debugging()

**提示**

开启函数必须在锁函数调用前调用。为了保障安全，你应该在设置完线程、锁函数后，第一时间开启。

最早出现在Libevent 2.0.4-alpha 中，叫 evthread\_enable\_lock\_debuging()。后面在2.1.2-alpha修改拼写，改为evthread\_enable\_lock\_debugging() 。现在这两个名字都支持。

**如何调试事件**

Libevent能够检查出来一些事件错误，并且报告给程序。包括:

* 使用未初始化的事件结构
* 尝试重新初始化事件结构

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 28 / 120 |

跟踪事件初始化需要额外的内存和CPU。所以你应该只在需要调试的时候才打开调试模式。

Interface

**void** event\_enable\_debug\_mode(**void**);

这个函数必须在任何event\_base被生成前调用。

在调试模式中，有大量被event\_assign()创建出来的事件时 [not event\_new()]，会使用大量的内存。这是因为Libevent不能判断事件是否还可用。(而event\_new()创建的事件可以通过event\_free()知道。) ，为了避免内存耗完，你可以显示地通知Libevent，事件已经释放。

Interface

**void** event\_debug\_unassign(**struct** event\*ev);

event\_debug\_unassign() 在非调试模式，没有任何作用

Example

#include <event2/event.h>

#include <event2/event\_struct.h>

#include <stdlib.h>

**void** cb(evutil\_socket\_t fd, **short** what, **void** \*ptr)

{

/\* We pass ’NULL’ as the callback pointer for the heap allocated

* event, and we pass the event itself as the callback pointer
* for the stack-allocated event. \*/

**struct** event\*ev = ptr;

**if** (ev)

event\_debug\_unassign(ev);

}

/\* Here’s a simple mainloop that waits until fd1 and fd2 are both \* ready to read. \*/

**void** mainloop(evutil\_socket\_t fd1, evutil\_socket\_t fd2, **int** debug\_mode)

{

**struct** event\_base\*base;

**struct** event event\_on\_stack,\*event\_on\_heap;

**if** (debug\_mode)

event\_enable\_debug\_mode();

base = event\_base\_new();

event\_on\_heap = event\_new(base, fd1, EV\_READ, cb, NULL);

event\_assign(&event\_on\_stack, base, fd2, EV\_READ, cb, &event\_on\_stack);

event\_add(event\_on\_heap, NULL);

event\_add(&event\_on\_stack, NULL);

event\_base\_dispatch(base);

event\_free(event\_on\_heap);

event\_base\_free(base);

}

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 29 / 120 |

调试模式需要在编译中设置"-DUSE\_DEBUG"。 链接了Libevent的程序会在后端输出大量的日志。这些日志包含且不限于以下这些:

* 增加事件
* 删除事件
* 操作系统相关通知信息

这些不能通过开关控制，只能在编译时设置。

首次在 Libevent 2.0.4-alpha中加入。

**检查Libevent版本**

新版本的Libevent加入了新特性，修正了Bug。有时候你需要知道Libevent的版本:

* Libevent版本是否合适你的程序。
* 为了调试，获取Libevent版本.
* 告诉用户Libevent版本，让他们注意相应版本的bug，更好地使用。

Interface

#define LIBEVENT\_VERSION\_NUMBER 0x02000300

#define LIBEVENT\_VERSION "2.0.3-alpha"

**const char** \*event\_get\_version(**void**);

ev\_uint32\_t event\_get\_version\_number(**void**);

宏是在编译时设置的。函数返回实时版本。如果你的程序动态链接Libevent，这些版本可能会不同。

你可以获取两种格式的版本信息: 字符串格式的，或4字节的整型。整型用最高字节表示高版本号， 第二字节表示小版本号，三字节表示补丁版本，最低字节表示是否发布版，0表示发布版，非0表示开发版。

比如Libevent 2.0.1-alpha 版本号是0x02000100，2.0.1-alpha和 2.0.2-alpha之间的开发版，版本号是 0x02000108。

Example: 编译时检查

#include <event2/event.h>

#**if** !defined(LIBEVENT\_VERSION\_NUMBER) || LIBEVENT\_VERSION\_NUMBER < 0x02000100 #error "This version of Libevent is not supported; Get 2.0.1-alpha or later." #endif

**int**

make\_sandwich(**void**)

{

/\* Let’s suppose that Libevent 6.0.5 introduces a make-me-a sandwich function. \*/

#**if** LIBEVENT\_VERSION\_NUMBER >= 0x06000500

evutil\_make\_me\_a\_sandwich();

**return** 0;

#**else**

**return** -1;

#endif

}

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 30 / 120 |

Example: 运行时检查

#include <event2/event.h>

#include <string.h>

**int**

check\_for\_old\_version(**void**)

{

**const char** \*v = event\_get\_version();

/\* This is a dumb way to do it, but it is the only thing that works before Libevent 2.0. \*/

**if** (!strncmp(v, "0.", 2) ||

!strncmp(v, "1.1", 3) ||

!strncmp(v, "1.2", 3) ||

!strncmp(v, "1.3", 3)) {

printf("Your version of Libevent is very old. If you run into bugs,"

* + consider upgrading.\n"); **return** -1;

} **else** {

printf("Running with Libevent version %s\n", v); **return** 0;

}

}

**int**

check\_version\_match(**void**)

{

ev\_uint32\_t v\_compile, v\_run;

v\_compile = LIBEVENT\_VERSION\_NUMBER;

v\_run = event\_get\_version\_number();

**if** ((v\_compile & 0xffff0000) != (v\_run & 0xffff0000)) {

printf("Running with a Libevent version (%s) very different from the "

"one we were built with (%s).\n", event\_get\_version(),

LIBEVENT\_VERSION);

**return** -1;

}

**return** 0;

}

这节中提到的宏和函数在<event2/event.h>。event\_get\_version()首次出现在Libevent 1.0c其它首次出现在 Libevent 2.0.1-alpha。

**释放libevent全局结构**

即使你释放了所有 Libevent分配的事件，还剩下一些全局的对象。这通常不是问题: 一旦程序退出，它们都会被释放。但有些检查工具可能会认为Libevent 导致了资源泄露。如果你需要释放这些对象， 你可以调用 :

Interface

**void** libevent\_global\_shutdown(**void**);

这个函数并不会释放所有Libevent函数分配的对象。如果你想在程序退出前释放所有对象，你需要先释放掉所有event、event\_bases、bufferevents等等。

调用 libevent\_global\_shutdown() 会导致其它Libevent函数行为不正确。这行代码必须是调用Libevent的最后一行代码。唯一的例外是libevent\_global\_shutdown()可以被调用，它可以被重复调用多次。

这个函数定义在 <event2/event.h>，它在 Libevent 2.1.1-alpha被引入。

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 31 / 120 |

**创建 event\_base**

在你使用任何感兴趣的Libevent 函数前，你需要分配一个或多个event\_base结构。每个event\_base结构管理多个可被检测的事件。

如果event\_base 在有锁环境中使用，多线程访问是安全的。它的loop只能在单个线程中执行。如果你想使用多线程，你可以为每个线程分配一个 event\_base。

**提示**

[在将来的版本中，Libevent可能会支持多线程访问]

每个event\_base检查事件是否触发的函数。这些函数是:

* select
* poll
* epoll
* kqueue
* devpoll
* evport
* win32

用户可以通过设置环境变量禁止掉这些。比如：如果你想禁止kqueue backend，设置EVENT\_NOKQUEUE环境变量。 如果你想在程序中禁止，请查看下方的event\_config\_avoid\_method() 。

**设置默认event\_base**

event\_base\_new() 返回一个默认的event\_base。它检查环境变量，找到合适的配置，生成event\_base。 如果发生错误，它将返回NULL。

当选择上面的函数时，libevent会选择操作系统支持的性能最好的函数。

Interface

**struct** event\_base\*event\_base\_new(**void**);

多数情况下，这就是你需要的。

event\_base\_new() 定义在 <event2/event.h>，它首次出现在 Libevent 1.4.3。

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 32 / 120 |

**设置复杂的event\_base**

如果你想更多的掌控event\_base，你必须配置event\_config。一个event\_config是一个隐藏的结构。它包括了event\_base的参数。使用event\_config 参数调用 event\_base\_new\_with\_config()，就会得到特定的event\_base。

Interface

**struct** event\_config\*event\_config\_new(**void**);

**struct** event\_base\*event\_base\_new\_with\_config(**const struct** event\_config\*cfg); **void** event\_config\_free(**struct** event\_config\*cfg);

要利用这个函数生成的event\_base，你可以调用event\_config\_new() 分配一个event\_config.。然后调用函数修改event\_config。最后调用event\_base\_new\_with\_config()去获取一个新的event\_base。完成后，你可以调用event\_config\_free()来释放event\_config 。

Interface

**int** event\_config\_avoid\_method(**struct** event\_config\*cfg, **const char** \*method);

**enum** event\_method\_feature {

EV\_FEATURE\_ET = 0x01,

EV\_FEATURE\_O1 = 0x02,

EV\_FEATURE\_FDS = 0x04,

};

**int** event\_config\_require\_features(**struct** event\_config\*cfg,

**enum** event\_method\_feature feature);

**enum** event\_base\_config\_flag {

EVENT\_BASE\_FLAG\_NOLOCK = 0x01,

EVENT\_BASE\_FLAG\_IGNORE\_ENV = 0x02,

EVENT\_BASE\_FLAG\_STARTUP\_IOCP = 0x04,

EVENT\_BASE\_FLAG\_NO\_CACHE\_TIME = 0x08,

EVENT\_BASE\_FLAG\_EPOLL\_USE\_CHANGELIST = 0x10,

EVENT\_BASE\_FLAG\_PRECISE\_TIMER = 0x20

};

**int** event\_config\_set\_flag(**struct** event\_config \*cfg, **enum** event\_base\_config\_flag flag);

调用event\_config\_avoid\_method告诉Libevent禁止指定名称的后端函数。 调用event\_config\_require\_feature() 告诉Libevent 禁止不能支持所有特性的后端。 调用event\_config\_set\_flag() 告诉Libevent 设置更多的参数。

event\_config\_require\_features 接受的参数包括：

EV\_FEATURE\_ET

支持边沿触发

EV\_FEATURE\_O1

时间复杂度为 O(1) 。

EV\_FEATURE\_FDS

支持其它fd，不仅仅是socket

event\_config\_set\_flag() 参数包括:

EVENT\_BASE\_FLAG\_NOLOCK

不使用锁，这样会加快速度，但在多线程中是不安全的。

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 33 / 120 |

EVENT\_BASE\_FLAG\_IGNORE\_ENV

在选择后端函数时不要考虑EVENT\_\* 环境变量。想好了再设置这个flag。:它将会让你的程序和Libevent变得很难调试。

EVENT\_BASE\_FLAG\_STARTUP\_IOCP

只在Windows有效，设置之后，启动时就会进行事件重。

EVENT\_BASE\_FLAG\_NO\_CACHE\_TIME

在每个回调函数调用后，查检定时器是否到期，这样会消耗更多的性能。EVENT\_BASE\_FLAG\_EPOLL\_USE\_CHANGELIST

告知Libevent，如果决定使用epoll这个多路IO复用函数，可以安全地使用更快的基于changelist 的多路IO复用函数：epoll-changelist多路IO复用可以在多路IO复用函数调用之间，同样的fd 多次修改其状态的情况下，避免不必要的系统调用。但是如果传递任何使用dup()或者其变体克隆的fd给libevent，epoll-changelist多路IO复用函数会触发一个内核bug，导致不正确的结果。在不使用epoll这个多路IO复用函数的情况下，这个标志是没有效果的。也可以通过设置EVENT\_EPOLL\_USE\_CHANGELIST 环境变量来打开epoll-changelist选项。

EVENT\_BASE\_FLAG\_PRECISE\_TIMER

By default, Libevent tries to use the fastest available timing mechanism that the operating system provides. If there is a slower timing mechanism that provides more fine-grained timing precision, this flag tells Libevent to use that timing mechanism instead. If the operating system provides no such slower-but-more-precise mechanism, this flag has no effect.

The above functions that manipulate an event\_config all return 0 on success, -1 on failure.

**Note**

It is easy to set up an event\_config that requires a backend that your OS does not provide. For example, as of Libevent 2.0.1-alpha, there is no O(1) backend for Windows, and no backend on Linux that provides both EV\_FEATURE\_FDS and EV\_FEATURE\_O1. If you have made a configuration that Libevent can’t satisfy, event\_base\_new\_with\_config() will return NULL.

Interface

**int** event\_config\_set\_num\_cpus\_hint(**struct** event\_config\*cfg, **int** cpus)

This function is currently only useful with Windows when using IOCP, though it may become useful for other platforms in the future. Calling it tells the event\_config that the event\_base it generates should try to make good use of a given number of CPUs when multithreading. Note that this is only a hint: the event base may wind up using more or fewer CPUs than you select.

Interface

**int** event\_config\_set\_max\_dispatch\_interval(**struct** event\_config\*cfg,

**const struct** timeval\*max\_interval, **int** max\_callbacks,

**int** min\_priority);

This function prevents priority inversion by limiting how many low-priority event callbacks can be invoked before checking for more high-priority events. If max\_interval is non-null, the event loop checks the time after each callback, and re-scans for high-priority events if max\_interval has passed. If max\_callbacks is nonnegative, the event loop also checks for more events after max\_callbacks callbacks have been invoked. These rules apply to any event of min\_priority or higher.

Example: Preferring edge-triggered backends

**struct** event\_config\*cfg;

**struct** event\_base\*base;

**int** i;

/\* My program wants to use edge-triggered events if at all possible. So I’ll try to get a base twice: Once insisting on edge-triggered IO, and once not. \*/

**for** (i=0; i<2; ++i) {

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 34 / 120 |

cfg = event\_config\_new();

/\* I don’t like select. \*/

event\_config\_avoid\_method(cfg, "select");

**if** (i == 0)

event\_config\_require\_features(cfg, EV\_FEATURE\_ET);

base = event\_base\_new\_with\_config(cfg);

event\_config\_free(cfg);

**if** (base)

**break**;

/\* If we get here, event\_base\_new\_with\_config() returned NULL. If this is the first time around the loop, we’ll try again without setting EV\_FEATURE\_ET. If this is the second time around the loop, we’ll give up. \*/

}

Example: Avoiding priority-inversion

**struct** event\_config\*cfg;

**struct** event\_base\*base;

cfg = event\_config\_new();

**if** (!cfg)

/\* Handle error \*/;

/\* I’m going to have events running at two priorities. I expect that some of my priority-1 events are going to have pretty slow callbacks, so I don’t want more than 100 msec to elapse (or 5 callbacks) before checking for priority-0 events. \*/

**struct** timeval msec\_100 = { 0, 100\*1000 };

event\_config\_set\_max\_dispatch\_interval(cfg, &msec\_100, 5, 1);

base = event\_base\_new\_with\_config(cfg);

**if** (!base)

/\* Handle error \*/;

event\_base\_priority\_init(base, 2);

These functions and types are declared in <event2/event.h>.

The EVENT\_BASE\_FLAG\_IGNORE\_ENV flag first appeared in Libevent 2.0.2-alpha. The EVENT\_BASE\_FLAG\_PRECISE\_TIMER flag first appeared in Libevent 2.1.2-alpha. The event\_config\_set\_num\_cpus\_hint() function was new in Libevent 2.0.7-rc, and event\_config\_set\_max\_dispatch\_interval() was new in 2.1.1-alpha. Everything else in this section first appeared in Libevent 2.0.1-alpha.

**Examining an event\_base’s backend method**

Sometimes you want to see which features are actually available in an event\_base, or which method it’s using.

Interface

**const char** \*\*event\_get\_supported\_methods(**void**);

The event\_get\_supported\_methods() function returns a pointer to an array of the names of the methods supported in this version of Libevent. The last element in the array is NULL.

Example

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 35 / 120 |

**int** i;

**const char** \*\*methods = event\_get\_supported\_methods();printf("Starting Libevent %s. Available methods are:\n",

event\_get\_version());

**for** (i=0; methods[i] != NULL; ++i) {

printf(" %s\n", methods[i]);

}

**Note**

This function returns a list of the methods that Libevent was compiled to support. It is possible that your operating system will not in fact support them all when Libevent tries to run. For example, you could be on a version of OSX where kqueue is too buggy to use.

Interface

**const char** \*event\_base\_get\_method(**const struct** event\_base\*base);

**enum** event\_method\_feature event\_base\_get\_features(**const struct** event\_base\*base);

The event\_base\_get\_method() call returns the name of the actual method in use by an event\_base. The event\_base\_get\_features() call returns a bitmask of the features that it supports.

Example

**struct** event\_base\*base;

**enum** event\_method\_feature f;

base = event\_base\_new();

**if** (!base) {

puts("Couldn’t get an event\_base!");

} **else** {

printf("Using Libevent with backend method %s.", event\_base\_get\_method(base));

f = event\_base\_get\_features(base);

**if** ((f & EV\_FEATURE\_ET))

printf(" Edge-triggered events are supported."); **if** ((f & EV\_FEATURE\_O1))

printf(" O(1) event notification is supported."); **if** ((f & EV\_FEATURE\_FDS))

printf(" All FD types are supported.");

puts("");

}

These functions are defined in <event2/event.h>. The event\_base\_get\_method() call was first available in Libevent 1.4.3. The others first appeared in Libevent 2.0.1-alpha.

**Deallocating an event\_base**

When you are finished with an event\_base, you can deallocate it with event\_base\_free().

Interface

**void** event\_base\_free(**struct** event\_base\*base);

Note that this function does not deallocate any of the events that are currently associated with the event\_base, or close any of their sockets, or free any of their pointers.

The event\_base\_free() function is defined in <event2/event.h>. It was first implemented in Libevent 1.2.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 36 / 120 |

**Setting priorities on an event\_base**

Libevent supports setting multiple priorities on an event. By default, though, an event\_base supports only a single priority level.

You can set the number of priorities on an event\_base by calling event\_base\_priority\_init().

Interface

**int** event\_base\_priority\_init(**struct** event\_base\*base, **int** n\_priorities);

This function returns 0 on success and -1 on failure. The base argument is the event\_base to modify, and n\_priorities is the number of priorities to support. It must be at least 1. The available priorities for new events will be numbered from 0 (most important) to n\_priorities-1 (least important).

There is a constant, EVENT\_MAX\_PRIORITIES, that sets the upper bound on the value of n\_priorities. It is an error to call this function with a higher value for n\_priorities.

**Note**

You **must** call this function before any events become active. It is best to call it immediately after creating the event\_base.

To find the number of priorities currently supported by a base, you can call event\_base\_getnpriorities().

Interface

**int** event\_base\_get\_npriorities(**struct** event\_base\*base);

The return value is equal to the number of priorities configured in the base. So if event\_base\_get\_npriorities() returns 3, then allowable priority values are 0, 1, and 2.

Example

For an example, see the documentation for event\_priority\_set below.

By default, all new events associated with this base will be initialized with priority equal to n\_priorities / 2.

The event\_base\_priority\_init function is defined in <event2/event.h>. It has been available since Libevent 1.0. The event\_base\_get\_nprio function was new in Libevent 2.1.1-alpha.

**Reinitializing an event\_base after fork()**

Not all event backends persist cleanly after a call to fork(). Thus, if your program uses fork() or a related system call in order to start a new process, and you want to continue using an event\_base after you have forked, you may need to reinitialize it.

Interface

**int** event\_reinit(**struct** event\_base\*base);

The function returns 0 on success, -1 on failure.

Example

**struct** event\_base\*base = event\_base\_new();

/\* ... add some events to the event\_base ... \*/

**if** (fork()) {

/\* In parent \*/

continue\_running\_parent(base); /\*...\*/

} **else** {

/\* In child \*/

event\_reinit(base);

continue\_running\_child(base); /\*...\*/

}

The event\_reinit() function is defined in <event2/event.h>. It was first available in Libevent 1.4.3-alpha.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 37 / 120 |

**Obsolete event\_base functions**

Older versions of Libevent relied pretty heavily on the idea of a "current" event\_base. The "current" event\_base was a global setting shared across all threads. If you forgot to specify which event\_base you wanted, you got the current one. Since event\_bases weren’t threadsafe, this could get pretty error-prone.

Instead of event\_base\_new(), there was:

Interface

**struct** event\_base\*event\_init(**void**);

This function worked like event\_base\_new(), and set the current base to the allocated base. There was no other way to change the current base.

Some of the event\_base functions in this section had variants that operated on the current base. These functions behaved as the current functions, except that they took no base argument.

|  |  |
| --- | --- |
| Current function | Obsolete current-base version |
| event\_base\_priority\_init() | event\_priority\_init() |
| event\_base\_get\_method() | event\_get\_method() |

**Working with an event loop**

**Running the loop**

Once you have an event\_base with some events registered (see the next section about how to create and register events), you will want Libevent to wait for events and alert you about them.

Interface

|  |  |  |  |
| --- | --- | --- | --- |
| #define EVLOOP\_ONCE | | | 0x01 |
|  | #define | EVLOOP\_NONBLOCK | 0x02 |
| #define | | EVLOOP\_NO\_EXIT\_ON\_EMPTY 0x04 | |

**int** event\_base\_loop(**struct** event\_base\*base, **int** flags);

By default, the event\_base\_loop() function runs an event\_base until there are no more events registered in it. To run the loop, it repeatedly checks whether any of the registered events has triggered (for example, if a read event’s file descriptor is ready to read, or if a timeout event’s timeout is ready to expire). Once this happens, it marks all triggered events as "active", and starts to run them.

You can change the behavior of event\_base\_loop() by setting one or more flags in its flags argument. If EVLOOP\_ONCE is set, then the loop will wait until some events become active, then run active events until there are no more to run, then return. If EVLOOP\_NONBLOCK is set, then the loop will not wait for events to trigger: it will only check whether any events are ready to trigger immediately, and run their callbacks if so.

Ordinarily, the loop will exit as soon as it has no pending or active events. You can override this behavior by passing the EVLOOP\_NO\_EXIT\_ON\_EMPTY flag---for example, if you’re going to be adding events from some other thread. If you

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 38 / 120 |

do set EVLOOP\_NO\_EXIT\_ON\_EMPTY, the loop will keep running until somebody calls event\_base\_loopbreak(), or calls event\_base\_loopexit(), or an error occurs.

When it is done, event\_base\_loop() returns 0 if it exited normally, -1 if it exited because of some unhandled error in the backend, and 1 if it exited because there were no more pending or active events.

To aid in understanding, here’s an approximate summary of the event\_base\_loop algorithm:

Pseudocode

**while** (any events are registered with the loop,

or EVLOOP\_NO\_EXIT\_ON\_EMPTY was set) {

**if** (EVLOOP\_NONBLOCK was set, or any events are already active)If any registered events have triggered, mark them active.

**else**

Wait until at least one event has triggered, and mark it active.

**for** (p = 0; p < n\_priorities; ++p) {

**if** (any event with priority of p is active) {

Run all active events with priority of p.

**break**; /\*Do not run any events of a less important priority\*/

}

}

**if** (EVLOOP\_ONCE was set or EVLOOP\_NONBLOCK was set)

**break**;

}

As a convenience, you can also call:

Interface

**int** event\_base\_dispatch(**struct** event\_base\*base);

The event\_base\_dispatch() call is the same as event\_base\_loop(), with no flags set. Thus, it keeps running until there are no more registered events or until event\_base\_loopbreak() or event\_base\_loopexit() is called.

These functions are defined in <event2/event.h>. They have existed since Libevent 1.0.

**Stopping the loop**

If you want an active event loop to stop running before all events are removed from it, you have two slightly different functions you can call.

Interface

**int** event\_base\_loopexit(**struct** event\_base\*base, **const struct** timeval\*tv);

**int** event\_base\_loopbreak(**struct** event\_base\*base);

The event\_base\_loopexit() function tells an event\_base to stop looping after a given time has elapsed. If the tv argument is NULL, the event\_base stops looping without a delay. If the event\_base is currently running callbacks for any active events, it will continue running them, and not exit until they have all been run.

The event\_base\_loopbreak() function tells the event\_base to exit its loop immediately. It differs from event\_base\_loopexit(base, NULL) in that if the event\_base is currently running callbacks for any active events, it will exit immediately after finishing the one it’s currently processing.

Note also that event\_base\_loopexit(base,NULL) and event\_base\_loopbreak(base) act differently when no event loop is running: loopexit schedules the next instance of the event loop to stop right after the next round of callbacks are run (as if it had been invoked with EVLOOP\_ONCE) whereas loopbreak only stops a currently running loop, and has no effect if the event loop isn’t running.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 39 / 120 |

Both of these methods return 0 on success and -1 on failure.

Example: Shut down immediately

#include <event2/event.h>

/\* Here’s a callback function that calls loopbreak \*/ **void** cb(**int** sock, **short** what, **void** \*arg){

**struct** event\_base\*base = arg;

event\_base\_loopbreak(base);

}

**void** main\_loop(**struct** event\_base\*base, evutil\_socket\_t watchdog\_fd)

{

**struct** event\*watchdog\_event;

/\* Construct a new event to trigger whenever there are any bytes to read from a watchdog socket. When that happens, we’ll call the cb function, which will make the loop exit immediately without running any other active events at all.

\*/

watchdog\_event = event\_new(base, watchdog\_fd, EV\_READ, cb, base);

event\_add(watchdog\_event, NULL);

event\_base\_dispatch(base);

}

Example: Run an event loop for 10 seconds, then exit.

#include <event2/event.h>

**void** run\_base\_with\_ticks(**struct** event\_base\*base)

{

**struct** timeval ten\_sec;

ten\_sec.tv\_sec = 10;

ten\_sec.tv\_usec = 0;

|  |  |  |
| --- | --- | --- |
| /\* Now we | run the event\_base | for a series of 10-second intervals, printing |
| "Tick" | after each. For a | much better way to implement a 10-second |
| timer, | see the section below about persistent timer events. \*/ | |
| **while** (1) | { |  |
| /\* This schedules an exit | | ten seconds from now. \*/ |
| event\_base\_loopexit(base, | | &ten\_sec); |

event\_base\_dispatch(base);

puts("Tick");

}

}

Sometimes you may want to tell whether your call to event\_base\_dispatch() or event\_base\_loop() exited normally, or because of a call to event\_base\_loopexit() or event\_base\_break(). You can use these functions to tell whether loopexit or break was called:

Interface

**int** event\_base\_got\_exit(**struct** event\_base\*base); **int** event\_base\_got\_break(**struct** event\_base\*base);

These two functions will return true if the loop was stopped with event\_base\_loopexit() or event\_base\_break() respectively, and false otherwise. Their values will be reset the next time you start the event loop.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 40 / 120 |

These functions are declared in <event2/event.h>. The event\_break\_loopexit() function was first implemented in Libevent 1.0c; event\_break\_loopbreak() was first implemented in Libevent 1.4.3.

**Re-checking for events**

Ordinarily, Libevent checks for events, then runs all the active events with the highest priority, then checks for events again, and so on. But sometimes you might want to stop Libevent right after the current callback has been run, and tell it to scan again. By analogy to event\_base\_loopbreak(), you can do this with the function event\_base\_loopcontinue().

Interface

**int** event\_base\_loopcontinue(**struct** event\_base\*);

Calling event\_base\_loopcontinue() has no effect if we aren’t currently running event callbacks.

This function was introduced in Libevent 2.1.2-alpha.

**Checking the internal time cache**

Sometimes you want to get an approximate view of the current time inside an event callback, and you want to get it without calling gettimeofday() yourself (presumably because your OS implements gettimeofday() as a syscall, and you’re trying to avoid syscall overhead).

From within a callback, you can ask Libevent for its view of the current time when it began executing this round of callbacks:

Interface

**int** event\_base\_gettimeofday\_cached(**struct** event\_base\*base,

**struct** timeval\*tv\_out);

The event\_base\_gettimeofday\_cached() function sets the value of its tv\_out argument to the cached time if the event\_base is currently executing callbacks. Otherwise, it calls evutil\_gettimeofday() for the actual current time. It returns 0 on success, and negative on failure.

Note that since the timeval is cached when Libevent starts running callbacks, it will be at least a little inaccurate. If your callbacks take a long time to run, it may be very inaccurate. To force an immediate cache update, you can call this function:

Interface

**int** event\_base\_update\_cache\_time(**struct** event\_base\*base);

It returns 0 on success and -1 on failure, and has no effect if the base was not running its event loop.

The event\_base\_gettimeofday\_cached() function was new in Libevent 2.0.4-alpha. Libevent 2.1.1-alpha added event\_base\_update\_cache

**Dumping the event\_base status**

Interface

**void** event\_base\_dump\_events(**struct** event\_base\*base, FILE\*f);

For help debugging your program (or debugging Libevent!) you might sometimes want a complete list of all events added in the event\_base and their status. Calling event\_base\_dump\_events() writes this list to the stdio file provided.

The list is meant to be human-readable; its format will change in future versions of Libevent.

This function was introduced in Libevent 2.0.1-alpha.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 41 / 120 |

**Running a function over every event in an event\_base**

Interface

**typedef int** (\*event\_base\_foreach\_event\_cb)(**const struct** event\_base\*, **const struct** event\*, **void** \*);

**int** event\_base\_foreach\_event(**struct** event\_base\*base,

event\_base\_foreach\_event\_cb fn,

**void** \*arg);

You can use event\_base\_foreach\_event() to iterate over every currently active or pending event associated with an event\_base().

The provided callback will be invoked exactly once per event, in an unspecified order. The third argument of event\_base\_foreach\_event() will be passed as the third argument to each invocation of the callback.

The callback function must return 0 to continue iteration, or some other integer to stop iterating. Whatever value the callback function finally returns will then be returned by event\_base\_foreach\_function().

Your callback function must not modify any of the events that it receives, or add or remove any events to the event base, or otherwise modify any event associated with the event base, or undefined behavior can occur, up to or including crashes and heap-smashing.

The event\_base lock will be held for the duration of the call to event\_base\_foreach\_event() — this will block other threads from doing anything useful with the event\_base, so make sure that your callback doesn’t take a long time.

This function was added in Libevent 2.1.2-alpha.

**Obsolete event loop functions**

As discussed above, older versions of Libevent APIs had a global notion of a "current" event\_base.

Some of the event loop functions in this section had variants that operated on the current base. These functions behaved as the current functions, except that they took no base argument.

|  |  |
| --- | --- |
| Current function | Obsolete current-base version |
| event\_base\_dispatch() | event\_dispatch() |
| event\_base\_loop() | event\_loop() |
| event\_base\_loopexit() | event\_loopexit() |
| event\_base\_loopbreak() | event\_loopbreak() |

**Note**

Because event\_base did not support locking before Libevent 2.0, these functions weren’t completely threadsafe: it was not permissible to call the \_loopbreak() or \_loopexit() functions from a thread other than the one executing the event loop.

**Working with events**

Libevent’s basic unit of operation is the event. Every event represents a set of conditions, including:

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 42 / 120 |

* A file descriptor being ready to read from or write to.
* A file descriptor becoming ready to read from or write to (Edge-triggered IO only).
* A timeout expiring.
* A signal occurring.
* A user-triggered event.

Events have similar lifecycles. Once you call a Libevent function to set up an event and associate it with an event base, it becomes initialized. At this point, you can add, which makes it pending in the base. When the event is pending, if the conditions that would trigger an event occur (e.g., its file descriptor changes state or its timeout expires), the event becomes active, and its (user-provided) callback function is run. If the event is configured persistent, it remains pending. If it is not persistent, it stops being pending when its callback runs. You can make a pending event non-pending by deleting it, and you can add a non-pending event to make it pending again.

**Constructing event objects**

To create a new event, use the event\_new() interface.

Interface

|  |  |  |
| --- | --- | --- |
| #define EV\_TIMEOUT | | 0x01 |
|  | #define EV\_READ | 0x02 |
| #define EV\_WRITE | | 0x04 |
| #define EV\_SIGNAL | | 0x08 |
| #define EV\_PERSIST | | 0x10 |
| #define EV\_ET | | 0x20 |

**typedef void** (\*event\_callback\_fn)(evutil\_socket\_t, **short**, **void** \*);

**struct** event\*event\_new(**struct** event\_base\*base, evutil\_socket\_t fd, **short** what, event\_callback\_fn cb,

**void** \*arg);

**void** event\_free(**struct** event\*event);

The event\_new() function tries to allocate and construct a new event for use with base. The what argument is a set of the flags listed above. (Their semantics are described below.) If fd is nonnegative, it is the file that we’ll observe for read or write events. When the event is active, Libevent will invoke the provided cb function, passing it as arguments: the file descriptor fd, a bitfield of all the events that triggered, and the value that was passed in for arg when the function was constructed.

On an internal error, or invalid arguments, event\_new() will return NULL.

All new events are initialized and non-pending. To make an event pending, call event\_add() (documented below).

To deallocate an event, call event\_free(). It is safe to call event\_free() on an event that is pending or active: doing so makes the event non-pending and inactive before deallocating it.

Example

#include <event2/event.h>

**void** cb\_func(evutil\_socket\_t fd, **short** what, **void** \*arg)

{

|  |  |  |
| --- | --- | --- |
| **const char** \*data = arg; | |  |
| printf("Got an event on | | socket %d:%s%s%s%s [%s]", |
| (**int**) fd, |  |  |
| (what&EV\_TIMEOUT) ? | | " timeout" : "", |
| (what&EV\_READ) | ? | " read" : "", |
| (what&EV\_WRITE) | ? | " write" : "", |
| (what&EV\_SIGNAL) | ? | " signal" : "", |
|  |  |  |

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 43 / 120 |

data);

}

**void** main\_loop(evutil\_socket\_t fd1, evutil\_socket\_t fd2)

{

**struct** event\*ev1,\*ev2;

**struct** timeval five\_seconds = {5,0};

**struct** event\_base\*base = event\_base\_new();

/\* The caller has already set up fd1, fd2 somehow, and make them nonblocking. \*/

ev1 = event\_new(base, fd1, EV\_TIMEOUT|EV\_READ|EV\_PERSIST, cb\_func, (**char**\*)"Reading event");

ev2 = event\_new(base, fd2, EV\_WRITE|EV\_PERSIST, cb\_func, (**char**\*)"Writing event");

event\_add(ev1, &five\_seconds);

event\_add(ev2, NULL);

event\_base\_dispatch(base);

}

The above functions are defined in <event2/event.h>, and first appeared in Libevent 2.0.1-alpha. The event\_callback\_fn type first appeared as a typedef in Libevent 2.0.4-alpha.

**The event flags**

EV\_TIMEOUT

This flag indicates an event that becomes active after a timeout elapses.

The EV\_TIMEOUT flag is ignored when constructing an event: you can either set a timeout when you add the event, or not. It is set in the ’what’ argument to the callback function when a timeout has occurred.

EV\_READ

This flag indicates an event that becomes active when the provided file descriptor is ready for reading.

EV\_WRITE

This flag indicates an event that becomes active when the provided file descriptor is ready for writing.

EV\_SIGNAL

Used to implement signal detection. See "Constructing signal events" below.

EV\_PERSIST

Indicates that the event is persistent. See "About Event Persistence" below.

EV\_ET

Indicates that the event should be edge-triggered, if the underlying event\_base backend supports edge-triggered events.

This affects the semantics of EV\_READ and EV\_WRITE.

Since Libevent 2.0.1-alpha, any number of events may be pending for the same conditions at the same time. For example, you may have two events that will become active if a given fd becomes ready to read. The order in which their callbacks are run is undefined.

These flags are defined in <event2/event.h>. All have existed since before Libevent 1.0, except for EV\_ET, which was introduced in Libevent 2.0.1-alpha.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 44 / 120 |

**About Event Persistence**

By default, whenever a pending event becomes active (because its fd is ready to read or write, or because its timeout expires), it becomes non-pending right before its callback is executed. Thus, if you want to make the event pending again, you can call event\_add() on it again from inside the callback function.

If the EV\_PERSIST flag is set on an event, however, the event is persistent. This means that event remains pending even when its callback is activated. If you want to make it non-pending from within its callback, you can call event\_del() on it.

The timeout on a persistent event resets whenever the event’s callback runs. Thus, if you have an event with flags EV\_READ|EV\_PERSIS and a timeout of five seconds, the event will become active:

* Whenever the socket is ready for reading.
* Whenever five seconds have passed since the event last became active.

**Creating an event as its own callback argument**

Frequently, you might want to create an event that receives itself as a callback argument. You can’t just pass a pointer to the event as an argument to event\_new(), though, because it does not exist yet. To solve this problem, you can use event\_self\_cbarg().

Interface

**void** \*event\_self\_cbarg();

The event\_self\_cbarg() function returns a "magic" pointer which, when passed as an event callback argument, tells event\_new() to create an event receiving itself as its callback argument.

Example

#include <event2/event.h>

**static int** n\_calls = 0;

**void** cb\_func(evutil\_socket\_t fd, **short** what, **void** \*arg)

{

**struct** event\*me = arg;

printf("cb\_func called %d times so far.\n", ++n\_calls);

**if** (n\_calls > 100)

event\_del(me);

}

**void** run(**struct** event\_base\*base)

{

**struct** timeval one\_sec = { 1, 0 };

**struct** event\*ev;

/\* We’re going to set up a repeating timer to get called called 100 times. \*/

ev = event\_new(base, -1, EV\_PERSIST, cb\_func, event\_self\_cbarg());

event\_add(ev, &one\_sec);

event\_base\_dispatch(base);

}

This function can also be used with event\_new(), evtimer\_new(), evsignal\_new(), event\_assign(), evtimer\_assign(), and evsig-nal\_assign(). It won’t work as a callback argument for non-events, however.

The event\_self\_cbarg() function was introduced in Libevent 2.1.1-alpha.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 45 / 120 |

**Timeout-only events**

As a convenience, there are a set of macros beginning with evtimer\_ that you can use in place of the event\_\* calls to allocate and manipulate pure-timeout events. Using these macros provides no benefit beyond improving the clarity of your code.

Interface

#define evtimer\_new(base, callback, arg) \

event\_new((base), -1, 0, (callback), (arg))

#define evtimer\_add(ev, tv) \

event\_add((ev),(tv))

#define evtimer\_del(ev) \

event\_del(ev)

#define evtimer\_pending(ev, tv\_out) \

event\_pending((ev), EV\_TIMEOUT, (tv\_out))

These macros have been present since Libevent 0.6, except for evtimer\_new(), which first appeared in Libevent 2.0.1-alpha.

**Constructing signal events**

Libevent can also watch for POSIX-style signals. To construct a handler for a signal, use:

Interface

#define evsignal\_new(base, signum, cb, arg) \

event\_new(base, signum, EV\_SIGNAL|EV\_PERSIST, cb, arg)

The arguments are as for event\_new, except that we provide a signal number instead of a file descriptor.

Example

**struct** event\*hup\_event;

**struct** event\_base\*base = event\_base\_new();

/\* call sighup\_function on a HUP signal \*/

hup\_event = evsignal\_new(base, SIGHUP, sighup\_function, NULL);

Note that signal callbacks are run in the event loop after the signal occurs, so it is safe for them to call functions that you are not supposed to call from a regular POSIX signal handler.



**Warning**

Don’t set a timeout on a signal event. It might not be supported. [FIXME: is this true?]

There are also a set of convenience macros you can use when working with signal events.

Interface

#define evsignal\_add(ev, tv) \

event\_add((ev),(tv))

#define evsignal\_del(ev) \

event\_del(ev)

#define evsignal\_pending(ev, what, tv\_out) \

event\_pending((ev), (what), (tv\_out))

The evsignal\_\* macros have been present since Libevent 2.0.1-alpha. Prior versions called them signal\_add(), signal\_del(), and so on.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 46 / 120 |

**Caveats when working with signals**

With current versions of Libevent, with most backends, only one event\_base per process at a time can be listening for signals. If you add signal events to two event\_bases at once ---even if the signals are different!--- only one event\_base will receive signals.

The kqueue backend does not have this limitation.

**Setting up events without heap-allocation**

For performance and other reasons, some people like to allocate events as a part of a larger structure. For each use of the event, this saves them:

* The memory allocator overhead for allocating a small object on the heap.
* The time overhead for dereferencing the pointer to the struct event.
* The time overhead from a possible additional cache miss if the event is not already in the cache.

Using this method risks breaking binary compatibility with other versions of of Libevent, which may have different sizes for the event structure.

These are very small costs, and do not matter for most applications. You should just stick to using event\_new() unless you know that you’re incurring a significant performance penalty for heap-allocating your events. Using event\_assign() can cause hard-to-diagnose errors with future versions of Libevent if they use a larger event structure than the one you’re building with.

Interface

**int** event\_assign(**struct** event \*event, **struct** event\_base \*base, evutil\_socket\_t fd, **short** what,

**void** (\*callback)(evutil\_socket\_t, **short**, **void** \*), **void** \*arg);

All the arguments of event\_assign() are as for event\_new(), except for the event argument, which must point to an uninitialized event. It returns 0 on success, and -1 on an internal error or bad arguments.

Example

#include <event2/event.h>

/\* Watch out! Including event\_struct.h means that your code will not

* be binary-compatible with future versions of Libevent. \*/ #include <event2/event\_struct.h>

#include <stdlib.h>

**struct** event\_pair {

evutil\_socket\_t fd;

**struct** event read\_event;

**struct** event write\_event;

};

**void** readcb(evutil\_socket\_t, **short**, **void** \*);

**void** writecb(evutil\_socket\_t, **short**, **void** \*);

**struct** event\_pair\*event\_pair\_new(**struct** event\_base\*base, evutil\_socket\_t fd){

**struct** event\_pair\*p = malloc(**sizeof**(**struct** event\_pair)); **if** (!p) **return** NULL;

p->fd = fd;

event\_assign(&p->read\_event, base, fd, EV\_READ|EV\_PERSIST, readcb, p); event\_assign(&p->write\_event, base, fd, EV\_WRITE|EV\_PERSIST, writecb, p);

**return** p;

}

You can also use event\_assign() to initialize stack-allocated or statically allocated events.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 47 / 120 |

WARNING Never call event\_assign() on an event that is already pending in an event base. Doing so can lead to extremely hard-to-diagnose errors. If the event is already initialized and pending, call event\_del() on it before you call event\_assign() on it again.

There are convenience macros you can use to event\_assign() a timeout-only or a signal event:

Interface

#define evtimer\_assign(event, base, callback, arg) \ event\_assign(event, base, -1, 0, callback, arg)

#define evsignal\_assign(event, base, signum, callback, arg) \

event\_assign(event, base, signum, EV\_SIGNAL|EV\_PERSIST, callback, arg)

If you need to use event\_assign() and retain binary compatibility with future versions of Libevent, you can ask the Libevent library to tell you at runtime how large a struct event should be:

Interface

size\_t event\_get\_struct\_event\_size(**void**);

This function returns the number of bytes you need to set aside for a struct event. As before, you should only be using this function if you know that heap-allocation is actually a significant problem in your program, since it can make your code much harder to read and write.

Note that event\_get\_struct\_event\_size() may in the future give you a value smaller than sizeof(struct event). If this happens, it means that any extra bytes at the end of struct event are only padding bytes reserved for use by a future version of Libevent.

Here’s the same example as above, but instead of relying on the size of struct event from event\_struct.h, we use event\_get\_struct\_size() to use the correct size at runtime.

Example

#include <event2/event.h>

#include <stdlib.h>

/\* When we allocate an event\_pair in memory, we’ll actually allocate

* more space at the end of the structure. We define some macros
* to make accessing those events less error-prone. \*/

**struct** event\_pair {

evutil\_socket\_t fd;

};

/\* Macro: yield the struct event ’offset’ bytes from the start of ’p’ \*/ #define EVENT\_AT\_OFFSET(p, offset) \

((**struct** event\*) ( ((**char**\*)(p)) + (offset) ))

/\* Macro: yield the read event of an event\_pair \*/ #define READEV\_PTR(pair) \

EVENT\_AT\_OFFSET((pair), **sizeof**(**struct** event\_pair))

/\* Macro: yield the write event of an event\_pair \*/

#define WRITEEV\_PTR(pair) \

EVENT\_AT\_OFFSET((pair), \

**sizeof**(**struct** event\_pair)+event\_get\_struct\_event\_size())

/\* Macro: yield the actual size to allocate for an event\_pair \*/ #define EVENT\_PAIR\_SIZE() \

(**sizeof**(**struct** event\_pair)+2\*event\_get\_struct\_event\_size())

**void** readcb(evutil\_socket\_t, **short**, **void** \*);

**void** writecb(evutil\_socket\_t, **short**, **void** \*);

**struct** event\_pair\*event\_pair\_new(**struct** event\_base\*base, evutil\_socket\_t fd)

{

**struct** event\_pair\*p = malloc(EVENT\_PAIR\_SIZE()); **if** (!p) **return** NULL;

p->fd = fd;

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 48 / 120 |

event\_assign(READEV\_PTR(p), base, fd, EV\_READ|EV\_PERSIST, readcb, p); event\_assign(WRITEEV\_PTR(p), base, fd, EV\_WRITE|EV\_PERSIST, writecb, p); **return** p;

}

The event\_assign() function defined in <event2/event.h>. It has existed since Libevent 2.0.1-alpha. It has returned an int since 2.0.3-alpha; previously, it returned void. The event\_get\_struct\_event\_size() function was introduced in Libevent 2.0.4-alpha. The event structure itself is defined in <event2/event\_struct.h>.

**Making events pending and non-pending**

Once you have constructed an event, it won’t actually do anything until you have made it pending by adding it. You do this with event\_add:

Interface

**int** event\_add(**struct** event\*ev, **const struct** timeval\*tv);

Calling event\_add on a non-pending event makes it pending in its configured base. The function returns 0 on success, and -1 on failure. If tv is NULL, the event is added with no timeout. Otherwise, tv is the size of the timeout in seconds and microseconds.

If you call event\_add() on an event that is already pending, it will leave it pending, and reschedule it with the provided timeout.

If the event is already pending, and you re-add it with the timeout NULL, event\_add() will have no effect.

**Note**

Do not set tv to the time at which you want the timeout to run. If you say "tv!tv\_sec = time(NULL)+10;" on 1 January 2010, your timeout will wait 40 years, not 10 seconds.

Interface

**int** event\_del(**struct** event\*ev);

Calling event\_del on an initialized event makes it non-pending and non-active. If the event was not pending or active, there is no effect. The return value is 0 on success, -1 on failure.

**Note**

If you delete an event after it becomes active but before its callback has a chance to execute, the callback will not be executed.

Interface

**int** event\_remove\_timer(**struct** event\*ev);

Finally, you can remove a pending event’s timeout completely without deleting its IO or signal components. If the event had no timeout pending, event\_remove\_timer() has no effect. If the event had only a timeout but no IO or signal component, event\_remove\_timer() has the same effect as event\_del(). The return value is 0 on success, -1 on failure.

These are defined in <event2/event.h>; event\_add() and event\_del() have existed since Libevent 0.1; event\_remove\_timer() was added in 2.1.2-alpha.

**Events with priorities**

When multiple events trigger at the same time, Libevent does not define any order with respect to when their callbacks will be executed. You can define some events as more important than others by using priorities.

As discussed in an earlier section, each event\_base has one or more priority values associated with it. Before adding an event to the event\_base, but after initializing it, you can set its priority.

Interface

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 49 / 120 |

**int** event\_priority\_set(**struct** event\*event, **int** priority);

The priority of the event is a number between 0 and the number of priorities in an event\_base, minus 1. The function returns 0 on success, and -1 on failure.

When multiple events of multiple priorities become active, the low-priority events are not run. Instead, Libevent runs the high priority events, then checks for events again. Only when no high-priority events are active are the low-priority events run.

Example

#include <event2/event.h>

**void** read\_cb(evutil\_socket\_t, **short**, **void** \*);

**void** write\_cb(evutil\_socket\_t, **short**, **void** \*);

**void** main\_loop(evutil\_socket\_t fd)

{

**struct** event\*important,\*unimportant;

**struct** event\_base\*base;

base = event\_base\_new();

event\_base\_priority\_init(base, 2);

/\* Now base has priority 0, and priority 1 \*/

important = event\_new(base, fd, EV\_WRITE|EV\_PERSIST, write\_cb, NULL);

unimportant = event\_new(base, fd, EV\_READ|EV\_PERSIST, read\_cb, NULL);

event\_priority\_set(important, 0);

event\_priority\_set(unimportant, 1);

/\* Now, whenever the fd is ready for writing, the write callback will happen before the read callback. The read callback won’t happen at all until the write callback is no longer active. \*/

}

When you do not set the priority for an event, the default is the number of queues in the event base, divided by 2.

This function is declared in <event2/event.h>. It has existed since Libevent 1.0.

**Inspecting event status**

Sometimes you want to tell whether an event has been added, and check what it refers to.

Interface

**int** event\_pending(**const struct** event\*ev, **short** what, **struct** timeval\*tv\_out);

#define event\_get\_signal(ev) /\* ... \*/

evutil\_socket\_t event\_get\_fd(**const struct** event \*ev);

**struct** event\_base\*event\_get\_base(**const struct** event\*ev);

**short** event\_get\_events(**const struct** event\*ev);

event\_callback\_fn event\_get\_callback(**const struct** event \*ev);

**void** \*event\_get\_callback\_arg(**const struct** event\*ev);

**int** event\_get\_priority(**const struct** event\*ev);

**void** event\_get\_assignment(**const struct** event\*event, **struct** event\_base\*\*base\_out,evutil\_socket\_t \*fd\_out,

**short** \*events\_out,

event\_callback\_fn \*callback\_out,

**void** \*\*arg\_out);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 50 / 120 |

The event\_pending function determines whether the given event is pending or active. If it is, and any of the flags EV\_READ, EV\_WRITE, EV\_SIGNAL, and EV\_TIMEOUT are set in the what argument, the function returns all of the flags that the event is currently pending or active on. If tv\_out is provided, and EV\_TIMEOUT is set in what, and the event is currently pending or active on a timeout, then tv\_out is set to hold the time when the event’s timeout will expire.

The event\_get\_fd() and event\_get\_signal() functions return the configured file descriptor or signal number for an event. The event\_get\_base() function returns its configured event\_base. The event\_get\_events() function returns the event flags (EV\_READ, EV\_WRITE, etc) of the event. The event\_get\_callback() and event\_get\_callback\_arg() functions return the callback function and argument pointer. The event\_get\_priority() function returns the event’s currently assigned priority.

The event\_get\_assignment() function copies all of the assigned fields of the event into the provided pointers. If any of the pointers is NULL, it is ignored.

Example

#include <event2/event.h>

#include <stdio.h>

/\* Change the callback and callback\_arg of ’ev’, which must not be \* pending. \*/

**int** replace\_callback(**struct** event \*ev, event\_callback\_fn new\_callback, **void** \*new\_callback\_arg)

{

**struct** event\_base\*base;

evutil\_socket\_t fd;

**short** events;

**int** pending;

pending = event\_pending(ev, EV\_READ|EV\_WRITE|EV\_SIGNAL|EV\_TIMEOUT, NULL);

**if** (pending) {

/\* We want to catch this here so that we do not re-assign a

* pending event. That would be very very bad. \*/ fprintf(stderr,

"Error! replace\_callback called on a pending event!\n"); **return** -1;

}

event\_get\_assignment(ev, &base, &fd, &events,

NULL /\* ignore old callback \*/ ,

NULL /\* ignore old callback argument \*/);

event\_assign(ev, base, fd, events, new\_callback, new\_callback\_arg); **return** 0;

}

These functions are declared in <event2/event.h>. The event\_pending() function has existed since Libevent 0.1. Libevent 2.0.1-alpha introduced event\_get\_fd() and event\_get\_signal(). Libevent 2.0.2-alpha introduced event\_get\_base(). Libevent 2.1.2-alpha added event\_get\_priority(). The others were new in Libevent 2.0.4-alpha.

**Finding the currently running event**

For debugging or other purposes, you can get a pointer to the currently running event.

Interface

**struct** event\*event\_base\_get\_running\_event(**struct** event\_base\*base);

Note that this function’s behavior is only defined when it’s called from within the provided event\_base’s loop. Calling it from another thread is not supported, and can cause undefined behavior.

This function is declared in <event2/event.h>. It was introduced in Libevent 2.1.1-alpha.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 51 / 120 |

**Configuring one-off events**

If you don’t need to add an event more than once, or delete it once it has been added, and it doesn’t have to be persistent, you can use event\_base\_once().

Interface

**int** event\_base\_once(**struct** event\_base\*, evutil\_socket\_t, **short**,

**void** (\*)(evutil\_socket\_t, **short**, **void** \*), **void** \*, **const struct** timeval\*);

This function’s interface is the same as event\_new(), except that it does not support EV\_SIGNAL or EV\_PERSIST. The scheduled event is inserted and run with the default priority. When the callback is finally done, Libevent frees the internal event structure itself. The return value is 0 on success, -1 on failure.

Events inserted with event\_base\_once cannot be deleted or manually activated: if you want to be able to cancel an event, create it with the regular event\_new() or event\_assign() interfaces.

Note also that at up to Libevent 2.0, if the event is never triggered, the internal memory used to hold it will never be freed. Starting in Libevent 2.1.2-alpha, these events are freed when the event\_base is freed, even if they haven’t activated, but still be aware: if there’s some storage associated with their callback arguments, that storage won’t be released unless your program has done something to track and release it.

**Manually activating an event**

Rarely, you may want to make an event active even though its conditions have not triggered.

Interface

**void** event\_active(**struct** event\*ev, **int** what, **short** ncalls);

This function makes an event ev become active with the flags what (a combination of EV\_READ, EV\_WRITE, and EV\_TIMEOUT).

The event does not need to have previously been pending, and activating it does not make it pending.

Warning: calling event\_active() recursively on the same event may result in resource exhaustion. The following snippet of code is an example of how event\_active can be used incorrectly.

Bad Example: making an infinite loop with event\_active()

**struct** event\*ev;

**static void** cb(**int** sock, **short** which, **void** \*arg) {

/\* Whoops: Calling event\_active on the same event unconditionally from within its callback means that no other events might not get run! \*/

event\_active(ev, EV\_WRITE, 0);

}

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

**struct** event\_base\*base = event\_base\_new();

ev = event\_new(base, -1, EV\_PERSIST | EV\_READ, cb, NULL);

event\_add(ev, NULL);

event\_active(ev, EV\_WRITE, 0);

event\_base\_loop(base, 0);

**return** 0;

}

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 52 / 120 |

This creates a situation where the event loop is only executed once and calls the function "cb" forever.

Example: Alternative solution to the above problem using timers

**struct** event\*ev;

**struct** timeval tv;

**static void** cb(**int** sock, **short** which, **void** \*arg) { **if** (!evtimer\_pending(ev, NULL)) {

event\_del(ev);

evtimer\_add(ev, &tv);

}

}

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

**struct** event\_base\*base = event\_base\_new();

tv.tv\_sec = 0;

tv.tv\_usec = 0;

ev = evtimer\_new(base, cb, NULL);

evtimer\_add(ev, &tv);

event\_base\_loop(base, 0);

**return** 0;

}

Example: Alternative solution to the above problem using event\_config\_set\_max\_dispatch\_interval()

**struct** event\*ev;

**static void** cb(**int** sock, **short** which, **void** \*arg) {event\_active(ev, EV\_WRITE, 0);

}

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

**struct** event\_config\*cfg = event\_config\_new();

/\* Run at most 16 callbacks before checking for other events. \*/ event\_config\_set\_max\_dispatch\_interval(cfg, NULL, 16, 0); **struct** event\_base\*base = event\_base\_new\_with\_config(cfg);

ev = event\_new(base, -1, EV\_PERSIST | EV\_READ, cb, NULL);

event\_add(ev, NULL);

event\_active(ev, EV\_WRITE, 0);

event\_base\_loop(base, 0);

**return** 0;

}

This function is defined in <event2/event.h>. It has existed since Libevent 0.3.

**Optimizing common timeouts**

Current versions of Libevent use a binary heap algorithm to keep track of pending events’ timeouts. A binary heap gives performance of order O(lg n) for adding and deleting each event timeout. This is optimal if you’re adding events with a randomly distributed set of timeout values, but not if you have a large number of events with the same timeout.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 53 / 120 |

For example, suppose you have ten thousand events, each of which should trigger its timeout five seconds after it was added. In a situation like this, you could get O(1) performance for each timeout by using a doubly-linked queue implementation.

Naturally, you wouldn’t want to use a queue for all of your timeout values, since a queue is only faster for constant timeout values. If some of the timeouts are more-or-less randomly distributed, then adding one of those timeouts to a queue would take O(n) time, which would be significantly worse than a binary heap.

Libevent lets you solve this by placing some of your timeouts in queues, and others in the binary heap. To do this, you ask Libevent for a special "common timeout" timeval, which you then use to add events having that timeval. If you have a very large number of events with a single common timeout, using this optimization should improve timeout performance.

Interface

**const struct** timeval\*event\_base\_init\_common\_timeout(

**struct** event\_base\*base, **const struct** timeval\*duration);

This function takes as its arguments an event\_base, and the duration of the common timeout to initialize. It returns a pointer to a special struct timeval that you can use to indicate that an event should be added to an O(1) queue rather than the O(lg n) heap. This special timeval can be copied or assigned freely in your code. It will only work with the specific base you used to construct it. Do not rely on its actual contents: Libevent uses them to tell itself which queue to use.

Example

#include <event2/event.h>

#include <string.h>

/\* We’re going to create a very large number of events on a given base,

* nearly all of which have a ten-second timeout. If initialize\_timeout
* is called, we’ll tell Libevent to add the ten-second ones to an O(1)
* queue. \*/

**struct** timeval ten\_seconds = { 10, 0 };

**void** initialize\_timeout(**struct** event\_base\*base)

{

**struct** timeval tv\_in = { 10, 0 };

**const struct** timeval\*tv\_out;

tv\_out = event\_base\_init\_common\_timeout(base, &tv\_in); memcpy(&ten\_seconds, tv\_out, **sizeof**(**struct** timeval));

}

**int** my\_event\_add(**struct** event\*ev, **const struct** timeval\*tv)

{

/\* Note that ev must have the same event\_base that we passed to initialize\_timeout \*/

**if** (tv && tv->tv\_sec == 10 && tv->tv\_usec == 0)

**return** event\_add(ev, &ten\_seconds);

**else**

**return** event\_add(ev, tv);

}

As with all optimization functions, you should avoid using the common\_timeout functionality unless you’re pretty sure that it matters for you.

This functionality was introduced in Libevent 2.0.4-alpha.

**Telling a good event apart from cleared memory**

Libevent provides functions that you can use to distinguish an initialized event from memory that has been cleared by setting it to 0 (for example, by allocating it with calloc() or clearing it with memset() or bzero()).

Interface

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 54 / 120 |

**int** event\_initialized(**const struct** event\*ev);

#define evsignal\_initialized(ev) event\_initialized(ev)

#define evtimer\_initialized(ev) event\_initialized(ev)

Warning These functions can’t reliably distinguish between an initialized event and a hunk of uninitialized memory. You should not use them unless you know that the memory in question is either cleared or initialized as an event.

Generally, you shouldn’t need to use these functions unless you’ve got a pretty specific application in mind. Events returned by event\_new() are always initialized.

Example

#include <event2/event.h>

#include <stdlib.h>

**struct** reader {

evutil\_socket\_t fd;

};

#define READER\_ACTUAL\_SIZE() \

(**sizeof**(**struct** reader) + \

event\_get\_struct\_event\_size())

#define READER\_EVENT\_PTR(r) \

((**struct** event \*) (((**char**\*)(r))+**sizeof**(**struct** reader)))

**struct** reader\*allocate\_reader(evutil\_socket\_t fd)

{

**struct** reader\*r = calloc(1, READER\_ACTUAL\_SIZE()); **if** (r)

r->fd = fd;

**return** r;

}

**void** readcb(evutil\_socket\_t, **short**, **void** \*);

**int** add\_reader(**struct** reader\*r, **struct** event\_base\*b)

{

**struct** event\*ev = READER\_EVENT\_PTR(r);

**if** (!event\_initialized(ev))

event\_assign(ev, b, r->fd, EV\_READ, readcb, r); **return** event\_add(ev, NULL);

}

The event\_initialized() function has been present since Libevent 0.3.

**Obsolete event manipulation functions**

Pre-2.0 versions of Libevent did not have event\_assign() or event\_new(). Instead, you had event\_set(), which associated the event with the "current" base. If you had more than one base, you needed to remember to call event\_base\_set() afterwards to make sure that the event was associated with the base you actually wanted to use.

Interface

**void** event\_set(**struct** event\*event, evutil\_socket\_t fd, **short** what, **void**(\*callback)(evutil\_socket\_t, **short**, **void** \*), **void** \*arg);

**int** event\_base\_set(**struct** event\_base\*base, **struct** event\*event);

The event\_set() function was like event\_assign(), except for its use of the current base. The event\_base\_set() function changes the base associated with an event.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 55 / 120 |

There were variants of event\_set() for dealing more conveniently with timers and signals: evtimer\_set() corresponded roughly to evtimer\_assign(), and evsignal\_set() corresponded roughly to evsignal\_assign().

Versions of Libevent before 2.0 used "signal\_" as the prefix for the signal-based variants of event\_set() and so on, rather than "evsignal\_". (That is, they had signal\_set(), signal\_add(), signal\_del(), signal\_pending(), and signal\_initialized().) Truly ancient versions of Libevent (before 0.6) used "timeout\_" instead of "evtimer\_". Thus, if you’re doing code archeology, you might see timeout\_add(), timeout\_del(), timeout\_initialized(), timeout\_set(), timeout\_pending(), and so on.

In place of the event\_get\_fd() and event\_get\_signal() functions, older versions of Libevent (before 2.0) used two macros called EVENT\_FD() and EVENT\_SIGNAL(). These macros inspected the event structure’s contents directly and so prevented binary compatibility between versions; in 2.0 and later they are just aliases for event\_get\_fd() and event\_get\_signal().

Since versions of Libevent before 2.0 did not have locking support, it wasn’t safe to call any of the functions that change an event’s state with respect to a base from outside the thread running the base. These include event\_add(), event\_del(), event\_active(), and event\_base\_once().

There was also an event\_once() function that played the role of event\_base\_once(), but used the current base.

The EV\_PERSIST flag did not interoperate sensibly with timeouts before Libevent 2.0. Instead resetting the timeout whenever the event was activated, the EV\_PERSIST flag did nothing with the timeout.

Libevent versions before 2.0 did not support having multiple events inserted at the same time with the same fd and the same READ/WRITE. In other words, only one event at a time could be waiting for read on each fd, and only one event at a time could be waiting for write on each fd.

**Helper functions and types for Libevent**

The <event2/util.h> header defines many functions that you might find helpful for implementing portable applications using Libevent. Libevent uses these types and functions internally.

**Basic types**

**evutil\_socket\_t**

Most everywhere except Windows, a socket is an int, and the operating system hands them out in numeric order. Using the Windows socket API, however, a socket is of type SOCKET, which is really a pointer-like OS handle, and the order you receive them is undefined. We define the evutil\_socket\_t type to be an integer that can hold the output of socket() or accept() without risking pointer truncation on Windows.

Definition

#ifdef WIN32

#define evutil\_socket\_t intptr\_t

#**else**

#define evutil\_socket\_t **int**

#endif

This type was introduced in Libevent 2.0.1-alpha.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 56 / 120 |

**Standard integer types**

Often you will find yourself on a C system that missed out on the 21st century and therefore does not implement the standard

C99 stdint.h header. For this situation, Libevent defines its own versions of the bit-width-specific integers from stdint.h:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Type | Width | Signed | Maximum | Minimum |  |
| ev\_uint64\_t | 64 | No | EV\_UINT64\_MAX0 | |  |
| ev\_int64\_t | 64 | Yes | EV\_INT64\_MAXEV\_INT64\_MIN | | |
| ev\_uint32\_t | 32 | No | EV\_UINT32\_MAX0 | |  |
| ev\_int32\_t | 32 | Yes | EV\_INT32\_MAXEV\_INT32\_MIN | | |
| ev\_uint16\_t | 16 | No | EV\_UINT16\_MAX0 | |  |
| ev\_int16\_t | 16 | Yes | EV\_INT16\_MAXEV\_INT16\_MIN | | |
| ev\_uint8\_t | 8 | No | EV\_UINT8\_MAX0 | |  |
| ev\_int8\_t | 8 | Yes | EV\_INT8\_MAX EV\_INT8\_MIN | |  |

As in the C99 standard, each type has exactly the specified width, in bits.

These types were introduced in Libevent 1.4.0-beta. The MAX/MIN constants first appeared in Libevent 2.0.4-alpha.

**Miscellaneous compatibility types**

The ev\_ssize\_t type is defined to ssize\_t (signed size\_t) on platforms that have one, and to a reasonable default on platforms that don’t. The largest possible value of ev\_ssize\_t is EV\_SSIZE\_MAX; the smallest is EV\_SSIZE\_MIN. (The largest possible value for size\_t is EV\_SIZE\_MAX, in case your platform doesn’t define a SIZE\_MAX for you.)

The ev\_off\_t type is used to represent offset into a file or a chunk of memory. It is defined to off\_t on platforms with a reasonable off\_t definition, and to ev\_int64\_t on Windows.

Some implementations of the sockets API provide a length type, socklen\_t, and some do not. The ev\_socklen\_t is defined to this type where it exists, and a reasonable default otherwise.

The ev\_intptr\_t type is a signed integer that is large enough to hold a pointer without loss of bits. The ev\_uintptr\_t type is an unsigned integer large enough to hold a pointer without loss of bits.

The ev\_ssize\_t type was added in Libevent 2.0.2-alpha. The ev\_socklen\_t type was new in Libevent 2.0.3-alpha. The ev\_intptr\_t and ev\_uintptr\_t types, and the EV\_SSIZE\_MAX/MIN macros, were added in Libevent 2.0.4-alpha. The ev\_off\_t type first appeared in Libevent 2.0.9-rc.

**Timer portability functions**

Not every platform defines the standard timeval manipulation functions, so we provide our own implementations.

Interface

#define evutil\_timeradd(tvp, uvp, vvp) /\* ... \*/

#define evutil\_timersub(tvp, uvp, vvp) /\* ... \*/

These macros add or subtract (respectively) their first two arguments, and stores the result in the third.

Interface

#define evutil\_timerclear(tvp) /\* ... \*/

#define evutil\_timerisset(tvp) /\* ... \*/

Clearing a timeval sets its value to zero. Checking whether it is set returns true if it is nonzero and false otherwise.

Interface

#define evutil\_timercmp(tvp, uvp, cmp)

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 57 / 120 |

The evutil\_timercmp macro compares two timevals, and yields true if they are in the relationship specified by the relational operator cmp. For example, evutil\_timercmp(t1, t2, () means, "Is t1 ( t2?" Note that unlike some operating systems’ versions, Libevent’s timercmp supports all the C relational operations (that is, <, >, ==, !=, (, and >=).

Interface

**int** evutil\_gettimeofday(**struct** timeval\*tv, **struct** timezone\*tz);

The evutil\_gettimeofday function sets tv to the current time. The tz argument is unused.

Example

**struct** timeval tv1, tv2, tv3;

/\* Set tv1 = 5.5 seconds \*/

tv1.tv\_sec = 5; tv1.tv\_usec = 500\*1000;

/\* Set tv2 = now \*/

evutil\_gettimeofday(&tv2, NULL);

/\* Set tv3 = 5.5 seconds in the future \*/

evutil\_timeradd(&tv1, &tv2, &tv3);

/\* all 3 should print true \*/

**if** (evutil\_timercmp(&tv1, &tv1, ==)) /\* == "If tv1 == tv1" \*/

puts("5.5 sec == 5.5 sec");

**if** (evutil\_timercmp(&tv3, &tv2, >=)) /\* == "If tv3 >= tv2" \*/

puts("The future is after the present.");

**if** (evutil\_timercmp(&tv1, &tv2, <)) /\* == "If tv1 < tv2" \*/

puts("It is no longer the past.");

These functions were introduced in Libevent 1.4.0-beta, except for evutil\_gettimeofday(), which was introduced in Libevent 2.0.

**Note**

It wasn’t safe to use ( or >= with timercmp before Libevent 1.4.4.

**Socket API compatibility**

This section exists because, for historical reasons, Windows has never really implemented the Berkeley sockets API in a nice compatible (and nicely compatible) way. Here are some functions you can use in order to pretend that it has.

Interface

**int** evutil\_closesocket(evutil\_socket\_t s);

#define EVUTIL\_CLOSESOCKET(s) evutil\_closesocket(s)

This function closes a socket. On Unix, it’s an alias for close(); on Windows, it calls closesocket(). (You can’t use close() on sockets on Windows, and nobody else defines a closesocket().)

The evutil\_closesocket function was introduced in Libevent 2.0.5-alpha. Before then, you needed to call the EVUTIL\_CLOSESOCKET macro.

Interface

#define EVUTIL\_SOCKET\_ERROR()

#define EVUTIL\_SET\_SOCKET\_ERROR(errcode)

#define evutil\_socket\_geterror(sock)

#define evutil\_socket\_error\_to\_string(errcode)

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 58 / 120 |

These macros access and manipulate socket error codes. EVUTIL\_SOCKET\_ERROR() returns the global error code for the last socket operation from this thread, and evutil\_socket\_geterror() does so for a particular socket. (Both are errno on Unix-like systems.) EVUTIL\_SET\_SOCKET\_ERROR() changes the current socket error code (like setting errno on Unix), and evutil\_socket\_error\_to\_string() returns a string representation of a given socket error code (like strerror() on Unix).

(We need these functions because Windows doesn’t use errno for errors from socket functions, but instead uses WSAGetLastEr-ror().)

Note that the Windows socket errors are not the same as the standard-C errors you would see in errno; watch out. Interface

**int** evutil\_make\_socket\_nonblocking(evutil\_socket\_t sock);

Even the call you need to do nonblocking IO on a socket is not portable to Windows. The evutil\_make\_socket\_nonblocking() function takes a new socket (from socket() or accept()) and turns it into a nonblocking socket. (It sets O\_NONBLOCK on Unix and FIONBIO on Windows.)

Interface

**int** evutil\_make\_listen\_socket\_reuseable(evutil\_socket\_t sock);

This function makes sure that the address used by a listener socket will be available to another socket immediately after the socket is closed. (It sets SO\_REUSEADDR on Unix and does nothing on Windows. You don’t want to use SO\_REUSEADDR on Windows; it means something different there.)

Interface

**int** evutil\_make\_socket\_closeonexec(evutil\_socket\_t sock);

This call tells the operating system that this socket should be closed if we ever call exec(). It sets the FD\_CLOEXEC flag on Unix, and does nothing on Windows.

Interface

**int** evutil\_socketpair(**int** family, **int** type, **int** protocol,

evutil\_socket\_t sv[2]);

This function behaves as the Unix socketpair() call: it makes two sockets that are connected with each other and can be used with ordinary socket IO calls. It stores the two sockets in sv[0] and sv[1], and returns 0 for success and -1 for failure.

On Windows, this only supports family AF\_INET, type SOCK\_STREAM, and protocol 0. Note that this can fail on some Windows hosts where firewall software has cleverly firewalled 127.0.0.1 to keep the host from talking to itself.

These functions were introduced in Libevent 1.4.0-beta, except for evutil\_make\_socket\_closeonexec(), which was new in Libevent 2.0.4-alpha.

**Portable string manipulation functions**

Interface

ev\_int64\_t evutil\_strtoll(**const char** \*s, **char** \*\*endptr, **int** base);

This function behaves as strtol, but handles 64-bit integers. On some platforms, it only supports Base 10. Interface

**int** evutil\_snprintf(**char** \*buf, size\_t buflen, **const char** \*format, ...);

**int** evutil\_vsnprintf(**char** \*buf, size\_t buflen, **const char** \*format, va\_list ap);

These snprintf-replacement functions behave as the standard snprintf and vsnprintf interfaces. They return the number of bytes that would have been written into the buffer had it been long enough, not counting the terminating NUL byte. (This behavior conforms to the C99 snprintf() standard, and is in contrast to the Windows \_snprintf(), which returns a negative number if the string would not fit in the buffer.)

The evutil\_strtoll() function has been in Libevent since 1.4.2-rc. These other functions first appeared in version 1.4.5.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 59 / 120 |

**Locale-independent string manipulation functions**

Sometimes, when implementing ASCII-based protocols, you want to manipulate strings according to ASCII’s notion of character type, regardless of your current locale. Libevent provides a few functions to help with this:

Interface

**int** evutil\_ascii\_strcasecmp(**const char** \*str1, **const char** \*str2);

**int** evutil\_ascii\_strncasecmp(**const char** \*str1, **const char** \*str2, size\_t n);

These functions behave as strcasecmp() and strncasecmp(), except that they always compare using the ASCII character set, regardless of the current locale. The evutil\_ascii\_str[n]casecmp() functions were first exposed in Libevent 2.0.3-alpha.

**IPv6 helper and portability functions**

Interface

**const char** \*evutil\_inet\_ntop(**int** af, **const void** \*src, **char** \*dst, size\_t len); **int** evutil\_inet\_pton(**int** af, **const char** \*src, **void** \*dst);

These functions behave as the standard inet\_ntop() and inet\_pton() functions for parsing and formatting IPv4 and IPv6 addresses, as specified in RFC3493. That is, to format an IPv4 address, you call evutil\_inet\_ntop() with af set to AF\_INET, src pointing to a struct in\_addr, and dst pointing to a character buffer of size len. For an IPv6 address, af is AF\_INET6 and src is a struct in6\_addr. To parse an IPv4 address, call evutil\_inet\_pton() with af set to AF\_INET or AF\_INET6, the string to parse in src, and dst pointing to an in\_addr or an in\_addr6 as appropriate.

The return value from evutil\_inet\_ntop() is NULL on failure and otherwise points to dst. The return value from evutil\_inet\_pton() is 0 on success and -1 on failure.

Interface

**int** evutil\_parse\_sockaddr\_port(**const char** \*str, **struct** sockaddr\*out,

**int** \*outlen);

This function parses an address from str and writes the result to out. The outlen argument must point to an integer holding the number of bytes available in out; it is altered to hold the number of bytes actually used. This function returns 0 on success and -1 on failure. It recognizes the following address formats:

* [ipv6]:port (as in "[ffff::]:80")
* ipv6 (as in "ffff::")
* [ipv6] (as in "[ffff::]")
* ipv4:port (as in "1.2.3.4:80")
* ipv4 (as in "1.2.3.4")

If no port is given, the port in the resulting sockaddr is set to 0.

Interface

**int** evutil\_sockaddr\_cmp(**const struct** sockaddr \*sa1, **const struct** sockaddr\*sa2, **int** include\_port);

The evutil\_sockaddr\_cmp() function compares two addresses, and returns negative if sa1 precedes sa2, 0 if they are equal, and positive if sa2 precedes sa1. It works for AF\_INET and AF\_INET6 addresses, and returns undefined output for other addresses. It’s guaranteed to give a total order for these addresses, but the ordering may change between Libevent versions.

If the include\_port argument is false, then two sockaddrs are treated as equal if they differ only in their port. Otherwise, sockaddrs with different ports are treated as unequal.

These functions were introduced in Libevent 2.0.1-alpha, except for evutil\_sockaddr\_cmp(), which introduced in 2.0.3-alpha.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 60 / 120 |

**Structure macro portability functions**

Interface

#define evutil\_offsetof(type, field) /\* ... \*/

As the standard offsetof macro, this macro yields the number of bytes from the start of type at which field occurs.

This macro was introduced in Libevent 2.0.1-alpha. It was buggy in every version before Libevent 2.0.3-alpha.

**Secure random number generator**

Many applications (including evdns) need a source of hard-to-predict random numbers for their security.

Interface

**void** evutil\_secure\_rng\_get\_bytes(**void** \*buf, size\_t n);

This function fills n-byte buffer at buf with n bytes of random data.

If your platform provides the arc4random() function, Libevent uses that. Otherwise, it uses its own implementation of arc4random(), seeded by your operating system’s entropy pool (CryptGenRandom on Windows, /dev/urandom everywhere else).

Interface

**int** evutil\_secure\_rng\_init(**void**);

**void** evutil\_secure\_rng\_add\_bytes(**const char** \*dat, size\_t datlen);

You do not need to manually initialize the secure random number generator, but if you want to make sure it is successfully initialized, you can do so by calling evutil\_secure\_rng\_init(). It seeds the RNG (if it was not already seeded) and returns 0 on success. If it returns -1, Libevent wasn’t able to find a good source of entropy on your OS, and you can’t use the RNG safely without initializing it yourself.

If you are running in an environment where your program is likely to drop privileges (for example, by running chroot()), you should call evutil\_secure\_rng\_init() before you do so.

You can add more random bytes to the entropy pool yourself by calling evutil\_secure\_rng\_add\_bytes(); this shouldn’t be neces-sary in typical use.

These functions are new in Libevent 2.0.4-alpha.

**Bufferevents: concepts and basics**

Most of the time, an application wants to perform some amount of data buffering in addition to just responding to events. When we want to write data, for example, the usual pattern runs something like:

* Decide that we want to write some data to a connection; put that data in a buffer.
* Wait for the connection to become writable

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 61 / 120 |

* Write as much of the data as we can
* Remember how much we wrote, and if we still have more data to write, wait for the connection to become writable again.

This buffered IO pattern is common enough that Libevent provides a generic mechanism for it. A "bufferevent" consists of an underlying transport (like a socket), a read buffer, and a write buffer. Instead of regular events, which give callbacks when the underlying transport is ready to be read or written, a bufferevent invokes its user-supplied callbacks when it has read or written enough data.

There are multiple types of bufferevent that all share a common interface. As of this writing, the following types exist:

socket-based bufferevents

A bufferevent that sends and receives data from an underlying stream socket, using the event\_\* interface as its backend.

asynchronous-IO bufferevents

A bufferevent that uses the Windows IOCP interface to send and receive data to an underlying stream socket. (Windows only; experimental.)

filtering bufferevents

A bufferevent that processes incoming and outgoing data before passing it to an underlying bufferevent object—for exam-ple, to compress or translate data.

paired bufferevents

Two bufferevents that transmit data to one another.

NOTE As of Libevent 2.0.2-alpha, the bufferevents interfaces here are still not fully orthogonal across all bufferevent types. In other words, not every interface described below will work on all bufferevent types. The Libevent developers intend to correct this in future versions.

NOTE ALSO Bufferevents currently only work for stream-oriented protocols like TCP. There may in the future be support for datagram-oriented protocols like UDP.

All of the functions and types in this section are declared in event2/bufferevent.h. Functions specifically related to evbuffers are declared in event2/buffer.h; see the next chapter for information on those.

**Bufferevents and evbuffers**

Every bufferevent has an input buffer and an output buffer. These are of type "struct evbuffer". When you have data to write on a bufferevent, you add it to the output buffer; when a bufferevent has data for you to read, you drain it from the input buffer.

The evbuffer interface supports many operations; we discuss them in a later section.

**Callbacks and watermarks**

Every bufferevent has two data-related callbacks: a read callback and a write callback. By default, the read callback is called whenever any data is read from the underlying transport, and the write callback is called whenever enough data from the output buffer is emptied to the underlying transport. You can override the behavior of these functions by adjusting the read and write "watermarks" of the bufferevent.

Every bufferevent has four watermarks:

Read low-water mark

Whenever a read occurs that leaves the bufferevent’s input buffer at this level or higher, the bufferevent’s read callback is invoked. Defaults to 0, so that every read results in the read callback being invoked.

Read high-water mark

If the bufferevent’s input buffer ever gets to this level, the bufferevent stops reading until enough data is drained from the input buffer to take us below it again. Defaults to unlimited, so that we never stop reading because of the size of the input buffer.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 62 / 120 |

Write low-water mark

Whenever a write occurs that takes us to this level or below, we invoke the write callback. Defaults to 0, so that a write callback is not invoked unless the output buffer is emptied.

Write high-water mark

Not used by a bufferevent directly, this watermark can have special meaning when a bufferevent is used as the underlying transport of another bufferevent. See notes on filtering bufferevents below.

A bufferevent also has an "error" or "event" callback that gets invoked to tell the application about non-data-oriented events, like when a connection is closed or an error occurs. The following event flags are defined:

BEV\_EVENT\_READING

An event occured during a read operation on the bufferevent. See the other flags for which event it was.

BEV\_EVENT\_WRITING

An event occured during a write operation on the bufferevent. See the other flags for which event it was.

BEV\_EVENT\_ERROR

An error occurred during a bufferevent operation. For more information on what the error was, call EVUTIL\_SOCKET\_ERROR().

BEV\_EVENT\_TIMEOUT

A timeout expired on the bufferevent.

BEV\_EVENT\_EOF

We got an end-of-file indication on the bufferevent.

BEV\_EVENT\_CONNECTED

We finished a requested connection on the bufferevent.

(The above event names are new in Libevent 2.0.2-alpha.)

**Deferred callbacks**

By default, a bufferevent callbacks are executed immediately when the corresponding condition happens. (This is true of evbuffer callbacks too; we’ll get to those later.) This immediate invocation can make trouble when dependencies get complex. For example, suppose that there is a callback that moves data into evbuffer A when it grows empty, and another callback that processes data out of evbuffer A when it grows full. Since these calls are all happening on the stack, you might risk a stack overflow if the dependency grows nasty enough.

To solve this, you can tell a bufferevent (or an evbuffer) that its callbacks should be deferred. When the conditions are met for a deferred callback, rather than invoking it immediately, it is queued as part of the event\_loop() call, and invoked after the regular events’ callbacks.

(Deferred callbacks were introduced in Libevent 2.0.1-alpha.)

**Option flags for bufferevents**

You can use one or more flags when creating a bufferevent to alter its behavior. Recognized flags are:

BEV\_OPT\_CLOSE\_ON\_FREE

When the bufferevent is freed, close the underlying transport. This will close an underlying socket, free an underlying bufferevent, etc.

BEV\_OPT\_THREADSAFE

Automatically allocate locks for the bufferevent, so that it’s safe to use from multiple threads.

BEV\_OPT\_DEFER\_CALLBACKS

When this flag is set, the bufferevent defers all of its callbacks, as described above.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 63 / 120 |

BEV\_OPT\_UNLOCK\_CALLBACKS

By default, when the bufferevent is set up to be threadsafe, the bufferevent’s locks are held whenever the any user-provided callback is invoked. Setting this option makes Libevent release the bufferevent’s lock when it’s invoking your callbacks.

(Libevent 2.0.5-beta introduced BEV\_OPT\_UNLOCK\_CALLBACKS. The other options above were new in Libevent 2.0.1-alpha.)

**Working with socket-based bufferevents**

The simplest bufferevents to work with is the socket-based type. A socket-based bufferevent uses Libevent’s underlying event mechanism to detect when an underlying network socket is ready for read and/or write operations, and uses underlying network calls (like readv, writev, WSASend, or WSARecv) to transmit and receive data.

**Creating a socket-based bufferevent**

You can create a socket-based bufferevent using bufferevent\_socket\_new():

Interface

**struct** bufferevent\*bufferevent\_socket\_new(

**struct** event\_base\*base,

evutil\_socket\_t fd,

**enum** bufferevent\_options options);

The base is an event\_base, and options is a bitmask of bufferevent options (BEV\_OPT\_CLOSE\_ON\_FREE, etc). The fd argu-ment is an optional file descriptor for a socket. You can set fd to -1 if you want to set the file descriptor later.

**Tip**

[Make sure that the socket you provide to bufferevent\_socket\_new is in non-blocking mode. Libevent provides the convenience method evutil\_make\_socket\_nonblocking for this.]

This function returns a bufferevent on success, and NULL on failure.

The bufferevent\_socket\_new() function was introduced in Libevent 2.0.1-alpha.

**Launching connections on socket-based bufferevents**

If the bufferevent’s socket is not yet connected, you can launch a new connection.

Interface

**int** bufferevent\_socket\_connect(**struct** bufferevent\*bev,

**struct** sockaddr\*address, **int** addrlen);

The address and addrlen arguments are as for the standard call connect(). If the bufferevent does not already have a socket set, calling this function allocates a new stream socket for it, and makes it nonblocking.

If the bufferevent does have a socket already, calling bufferevent\_socket\_connect() tells Libevent that the socket is not connected, and no reads or writes should be done on the socket until the connect operation has succeeded.

It is okay to add data to the output buffer before the connect is done.

This function returns 0 if the connect was successfully launched, and -1 if an error occurred.

Example

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 64 / 120 |

#include <event2/event.h>

#include <event2/bufferevent.h>

#include <sys/socket.h>

#include <string.h>

**void** eventcb(**struct** bufferevent\*bev, **short** events, **void** \*ptr)

{

**if** (events & BEV\_EVENT\_CONNECTED) {

/\* We’re connected to 127.0.0.1:8080. Ordinarily we’d do something here, like start reading or writing. \*/

} **else if** (events & BEV\_EVENT\_ERROR) {

/\* An error occured while connecting. \*/

}

}

**int** main\_loop(**void**)

{

**struct** event\_base\*base;

**struct** bufferevent\*bev;

**struct** sockaddr\_in sin;

base = event\_base\_new();

memset(&sin, 0, **sizeof**(sin));

sin.sin\_family = AF\_INET;

sin.sin\_addr.s\_addr = htonl(0x7f000001); /\* 127.0.0.1 \*/ sin.sin\_port = htons(8080); /\* Port 8080 \*/

bev = bufferevent\_socket\_new(base, -1, BEV\_OPT\_CLOSE\_ON\_FREE);

bufferevent\_setcb(bev, NULL, NULL, eventcb, NULL);

**if** (bufferevent\_socket\_connect(bev,

(**struct** sockaddr \*)&sin, **sizeof**(sin)) < 0) {

/\* Error starting connection \*/

bufferevent\_free(bev);

**return** -1;

}

event\_base\_dispatch(base);

**return** 0;

}

The bufferevent\_socket\_connect() function was introduced in Libevent-2.0.2-alpha. Before then, you had to manually call con-nect() on your socket yourself, and when the connection was complete, the bufferevent would report it as a write.

Note that you only get a BEV\_EVENT\_CONNECTED event if you launch the connect() attempt using bufferevent\_socket\_connect().

If you call connect() on your own, the connection gets reported as a write.

If you want to call connect() yourself, but still get receive a BEV\_EVENT\_CONNECTED event when the connection succeeds, call bufferevent\_socket\_connect(bev, NULL, 0) after connect() returns -1 with errno equal to EAGAIN or EINPROGRESS.

This function was introduced in Libevent 2.0.2-alpha.

**Launching connections by hostname**

Quite often, you’d like to combine resolving a hostname and connecting to it into a single operation. There’s an interface for that:

Interface

**int** bufferevent\_socket\_connect\_hostname(**struct** bufferevent\*bev,

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 65 / 120 |

**struct** evdns\_base\*dns\_base, **int** family, **const char** \*hostname, **int** port);

**int** bufferevent\_socket\_get\_dns\_error(**struct** bufferevent\*bev);

This function resolves the DNS name hostname, looking for addresses of type family. (Allowable family types are AF\_INET, AF\_INET6, and AF\_UNSPEC.) If the name resolution fails, it invokes the event callback with an error event. If it succeeds, it launches a connection attempt just as bufferevent\_connect would.

The dns\_base argument is optional. If it is NULL, then Libevent blocks while waiting for the name lookup to finish, which usually isn’t what you want. If it is provided, then Libevent uses it to look up the hostname asynchronously. See [chapter R9](Ref9_dns.html) for more info on DNS.

As with bufferevent\_socket\_connect(), this function tells Libevent that any existing socket on the bufferevent is not connected, and no reads or writes should be done on the socket until the resolve is finished and the connect operation has succeeded.

If an error occurs, it might be a DNS hostname lookup error. You can find out what the most recent error was by calling bufferevent\_socket\_get\_dns\_error(). If the returned error code is 0, no DNS error was detected.

Example: Trivial HTTP v0 client.

/\* Don’t actually copy this code: it is a poor way to implement an HTTP client. Have a look at evhttp instead.

\*/

#include <event2/dns.h>

#include <event2/bufferevent.h>

#include <event2/buffer.h>

#include <event2/util.h>

#include <event2/event.h>

#include <stdio.h>

**void** readcb(**struct** bufferevent\*bev, **void** \*ptr)

{

**char** buf[1024];

**int** n;

**struct** evbuffer\*input = bufferevent\_get\_input(bev);

**while** ((n = evbuffer\_remove(input, buf, **sizeof**(buf))) > 0) {fwrite(buf, 1, n, stdout);

}

}

**void** eventcb(**struct** bufferevent\*bev, **short** events, **void** \*ptr)

{

**if** (events & BEV\_EVENT\_CONNECTED) {

printf("Connect okay.\n");

} **else if** (events & (BEV\_EVENT\_ERROR|BEV\_EVENT\_EOF)) { **struct** event\_base\*base = ptr;

**if** (events & BEV\_EVENT\_ERROR) {

**int** err = bufferevent\_socket\_get\_dns\_error(bev); **if** (err)

printf("DNS error: %s\n", evutil\_gai\_strerror(err));

}

printf("Closing\n");

bufferevent\_free(bev);

event\_base\_loopexit(base, NULL);

}

}

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

{

**struct** event\_base\*base;

**struct** evdns\_base\*dns\_base;

**struct** bufferevent\*bev;

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 66 / 120 |

**if** (argc != 3) {

printf("Trivial HTTP 0.x client\n"

"Syntax: %s [hostname] [resource]\n"

"Example: %s www.google.com /\n",argv[0],argv[0]); **return** 1;

}

base = event\_base\_new();

dns\_base = evdns\_base\_new(base, 1);

bev = bufferevent\_socket\_new(base, -1, BEV\_OPT\_CLOSE\_ON\_FREE);

bufferevent\_setcb(bev, readcb, NULL, eventcb, base);

bufferevent\_enable(bev, EV\_READ|EV\_WRITE);

evbuffer\_add\_printf(bufferevent\_get\_output(bev), "GET %s\r\n", argv[2]);

bufferevent\_socket\_connect\_hostname(

bev, dns\_base, AF\_UNSPEC, argv[1], 80);

event\_base\_dispatch(base);

**return** 0;

}

The bufferevent\_socket\_connect\_hostname() function was new in Libevent 2.0.3-alpha; bufferevent\_socket\_get\_dns\_error() was new in 2.0.5-beta.

**Generic bufferevent operations**

The functions in this section work with multiple bufferevent implementations.

**Freeing a bufferevent**

Interface

**void** bufferevent\_free(**struct** bufferevent\*bev);

This function frees a bufferevent. Bufferevents are internally reference-counted, so if the bufferevent has pending deferred callbacks when you free it, it won’t be deleted until the callbacks are done.

The bufferevent\_free() function does, however, try to free the bufferevent as soon as possible. If there is pending data to write on the bufferevent, it probably won’t be flushed before the bufferevent is freed.

If the BEV\_OPT\_CLOSE\_ON\_FREE flag was set, and this bufferevent has a socket or underlying bufferevent associated with it as its transport, that transport is closed when you free the bufferevent.

This function was introduced in Libevent 0.8.

**Manipulating callbacks, watermarks, and enabled operations**

Interface

**typedef void** (\*bufferevent\_data\_cb)(**struct** bufferevent\*bev, **void** \*ctx); **typedef void** (\*bufferevent\_event\_cb)(**struct** bufferevent\*bev,

**short** events, **void** \*ctx);

**void** bufferevent\_setcb(**struct** bufferevent\*bufev,bufferevent\_data\_cb readcb, bufferevent\_data\_cb writecb, bufferevent\_event\_cb eventcb, **void** \*cbarg);

**void** bufferevent\_getcb(**struct** bufferevent\*bufev,bufferevent\_data\_cb \*readcb\_ptr,

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 67 / 120 |

bufferevent\_data\_cb \*writecb\_ptr,

bufferevent\_event\_cb \*eventcb\_ptr,

**void** \*\*cbarg\_ptr);

The bufferevent\_setcb() function changes one or more of the callbacks of a bufferevent. The readcb, writecb, and eventcb functions are called (respectively) when enough data is read, when enough data is written, or when an event occurs. The first argument of each is the bufferevent that has had the event happen. The last argument is the value provided by the user in the cbarg parameter of bufferevent\_callcb(): You can use this to pass data to your callbacks. The events argument of the event callback is a bitmask of event flags: see "callbacks and watermarks" above.

You can disable a callback by passing NULL instead of the callback function. Note all the callback functions on a bufferevent share a single cbarg value, so changing it will affect all of them.

You can retrieve the currently set callbacks for a bufferevent by passing pointers to bufferevent\_getcb(), which sets \*readcb\_ptr to the current read callback, \*writecb\_ptr to the current write callback, \*eventcb\_ptr to the current event callback, and \*cbarg\_ptr to the current callback argument field. Any of these pointers set to NULL will be ignored.

The bufferevent\_setcb() function was introduced in Libevent 1.4.4. The type names "bufferevent\_data\_cb" and "bufferevent\_event\_cb" were new in Libevent 2.0.2-alpha. The bufferevent\_getcb() function was added in 2.1.1-alpha.

Interface

**void** bufferevent\_enable(**struct** bufferevent\*bufev, **short** events); **void** bufferevent\_disable(**struct** bufferevent\*bufev, **short** events);

**short** bufferevent\_get\_enabled(**struct** bufferevent\*bufev);

You can enable or disable the events EV\_READ, EV\_WRITE, or EV\_READ|EV\_WRITE on a bufferevent. When reading or writing is not enabled, the bufferevent will not try to read or write data.

There is no need to disable writing when the output buffer is empty: the bufferevent automatically stops writing, and restarts again when there is data to write.

Similarly, there is no need to disable reading when the input buffer is up to its high-water mark: the bufferevent automatically stops reading, and restarts again when there is space to read.

By default, a newly created bufferevent has writing enabled, but not reading.

You can call bufferevent\_get\_enabled() to see which events are currently enabled on the bufferevent.

These functions were introduced in Libevent 0.8, except for bufferevent\_get\_enabled(), which was introduced in version 2.0.3-alpha.

Interface

**void** bufferevent\_setwatermark(**struct** bufferevent\*bufev, **short** events,

size\_t lowmark, size\_t highmark);

The bufferevent\_setwatermark() function adjusts the read watermarks, the write watermarks, or both, of a single bufferevent. (If EV\_READ is set in the events field, the read watermarks are adjusted. If EV\_WRITE is set in the events field, the write watermarks are adjusted.)

A high-water mark of 0 is equivalent to "unlimited".

This function was first exposed in Libevent 1.4.4.

Example

#include <event2/event.h>

#include <event2/bufferevent.h>

#include <event2/buffer.h>

#include <event2/util.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 68 / 120 |

**struct** info {

**const char** \*name;

size\_t total\_drained;

};

**void** read\_callback(**struct** bufferevent\*bev, **void** \*ctx)

{

**struct** info\*inf = ctx;

**struct** evbuffer\*input = bufferevent\_get\_input(bev);size\_t len = evbuffer\_get\_length(input);

**if** (len) {

inf->total\_drained += len;

evbuffer\_drain(input, len);

printf("Drained %lu bytes from %s\n",

(**unsigned long**) len, inf->name);

}

}

**void** event\_callback(**struct** bufferevent\*bev, **short** events, **void** \*ctx)

{

**struct** info\*inf = ctx;

**struct** evbuffer\*input = bufferevent\_get\_input(bev); **int** finished = 0;

**if** (events & BEV\_EVENT\_EOF) {

size\_t len = evbuffer\_get\_length(input);

printf("Got a close from %s. We drained %lu bytes from it, "

"and have %lu left.\n", inf->name,

(**unsigned long**)inf->total\_drained, (**unsigned long**)len); finished = 1;

}

**if** (events & BEV\_EVENT\_ERROR) {

printf("Got an error from %s: %s\n",

inf->name, evutil\_socket\_error\_to\_string(EVUTIL\_SOCKET\_ERROR())); finished = 1;

}

**if** (finished) {

free(ctx);

bufferevent\_free(bev);

}

}

**struct** bufferevent\*setup\_bufferevent(**void**)

{

**struct** bufferevent\*b1 = NULL;

**struct** info\*info1;

info1 = malloc(**sizeof**(**struct** info));

info1->name = "buffer 1";

info1->total\_drained = 0;

/\* ... Here we should set up the bufferevent and make sure it gets connected... \*/

/\* Trigger the read callback only whenever there is at least 128 bytes of data in the buffer. \*/

bufferevent\_setwatermark(b1, EV\_READ, 128, 0);

bufferevent\_setcb(b1, read\_callback, NULL, event\_callback, info1);

bufferevent\_enable(b1, EV\_READ); /\* Start reading. \*/

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 69 / 120 |

**return** b1;

}

**Manipulating data in a bufferevent**

Reading and writing data from the network does you no good if you can’t look at it. Bufferevents give you these methods to give them data to write, and to get the data to read:

Interface

**struct** evbuffer\*bufferevent\_get\_input(**struct** bufferevent\*bufev); **struct** evbuffer\*bufferevent\_get\_output(**struct** bufferevent\*bufev);

These two functions are very powerful fundamental: they return the input and output buffers respectively. For full information on all the operations you can perform on an evbuffer type, see the next chapter.

Note that the application may only remove (not add) data on the input buffer, and may only add (not remove) data from the output buffer.

If writing on the bufferevent was stalled because of too little data (or if reading was stalled because of too much), then adding data to the output buffer (or removing data from the input buffer) will automatically restart it.

These functions were introduced in Libevent 2.0.1-alpha.

Interface

**int** bufferevent\_write(**struct** bufferevent\*bufev,

**const void** \*data, size\_t size);

**int** bufferevent\_write\_buffer(**struct** bufferevent \*bufev, **struct** evbuffer\*buf);

These functions add data to a bufferevent’s output buffer. Calling bufferevent\_write() adds size bytes from the memory at data to the end of the output buffer. Calling bufferevent\_write\_buffer() removes the entire contents of buf and puts them at the end of the output buffer. Both return 0 if successful, or -1 if an error occurred.

These functions have existed since Libevent 0.8.

Interface

size\_t bufferevent\_read(**struct** bufferevent \*bufev, **void** \*data, size\_t size); **int** bufferevent\_read\_buffer(**struct** bufferevent\*bufev,

**struct** evbuffer\*buf);

These functions remove data from a bufferevent’s input buffer. The bufferevent\_read() function removes up to size bytes from the input buffer, storing them into the memory at data. It returns the number of bytes actually removed. The buffer-event\_read\_buffer() function drains the entire contents of the input buffer and places them into buf ; it returns 0 on success and -1 on failure.

Note that with bufferevent\_read(), the memory chunk at data must actually have enough space to hold size bytes of data.

The bufferevent\_read() function has existed since Libevent 0.8; bufferevent\_read\_buffer() was introduced in Libevent 2.0.1-alpha.

Example

#include <event2/bufferevent.h>

#include <event2/buffer.h>

#include <ctype.h>

**void**

read\_callback\_uppercase(**struct** bufferevent \*bev, **void** \*ctx)

{

/\* This callback removes the data from bev’s input buffer 128

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 70 / 120 |

bytes at a time, uppercases it, and starts sending it back.

(Watch out! In practice, you shouldn’t use toupper to implement a network protocol, unless you know for a fact that the current locale is the one you want to be using.)

\*/

**char** tmp[128];

size\_t n;

**int** i;

**while** (1) {

n = bufferevent\_read(bev, tmp, **sizeof**(tmp));

**if** (n <= 0)

**break**; /\*No more data.\*/

**for** (i=0; i<n; ++i)

tmp[i] = toupper(tmp[i]);

bufferevent\_write(bev, tmp, n);

}

}

**struct** proxy\_info {

**struct** bufferevent\*other\_bev;

};

**void**

read\_callback\_proxy(**struct** bufferevent \*bev, **void** \*ctx)

{

/\* You might use a function like this if you’re implementing a simple proxy: it will take data from one connection (on bev), and write it to another, copying as little as possible. \*/

**struct** proxy\_info\*inf = ctx;

bufferevent\_read\_buffer(bev,

bufferevent\_get\_output(inf->other\_bev));

}

**struct** count {

**unsigned long** last\_fib[2];

};

**void**

write\_callback\_fibonacci(**struct** bufferevent \*bev, **void** \*ctx)

{

/\* Here’s a callback that adds some Fibonacci numbers to the output buffer of bev. It stops once we have added 1k of data; once this data is drained, we’ll add more. \*/

**struct** count\*c = ctx;

**struct** evbuffer\*tmp = evbuffer\_new();

**while** (evbuffer\_get\_length(tmp) < 1024) {

**unsigned long** next = c->last\_fib[0] + c->last\_fib[1];c->last\_fib[0] = c->last\_fib[1]; c->last\_fib[1] = next;

evbuffer\_add\_printf(tmp, "%lu", next);

}

/\* Now we add the whole contents of tmp to bev. \*/ bufferevent\_write\_buffer(bev, tmp);

/\* We don’t need tmp any longer. \*/

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 71 / 120 |

evbuffer\_free(tmp);

}

**Read- and write timeouts**

As with other events, you can have a timeout get invoked if a certain amount of time passes without any data having been successfully written or read by a bufferevent.

Interface

**void** bufferevent\_set\_timeouts(**struct** bufferevent\*bufev,

**const struct** timeval\*timeout\_read, **const struct** timeval\*timeout\_write);

Setting a timeout to NULL is supposed to remove it; however before Libevent 2.1.2-alpha this wouldn’t work with all event types. (As a workaround for older versions, you can try setting the timeout to a multi-day interval and/or having your eventcb function ignore BEV\_TIMEOUT events when you don’t want them.)

The read timeout will trigger if the bufferevent waits at least timeout\_read seconds while trying to read read. The write timeout will trigger if the bufferevent waits at least timeout\_write seconds while trying to write data.

Note that the timeouts only count when the bufferevent would like to read or write. In other words, the read timeout is not enabled if reading is disabled on the bufferevent, or if the input buffer is full (at its high-water mark). Similarly, the write timeout is not enabled if if writing is disabled, or if there is no data to write.

When a read or write timeout occurs, the corresponding read or write operation becomes disabled on the bufferevent. The event callback is then invoked with either BEV\_EVENT\_TIMEOUT|BEV\_EVENT\_READING or BEV\_EVENT\_TIMEOUT|BEV\_EVENT\_

This functions has existed since Libevent 2.0.1-alpha. It didn’t behave consistently across bufferevent types until Libevent 2.0.4-alpha.

**Initiating a flush on a bufferevent**

Interface

**int** bufferevent\_flush(**struct** bufferevent \*bufev, **short** iotype, **enum** bufferevent\_flush\_mode state);

Flushing a bufferevent tells the bufferevent to force as many bytes as possible to be read to or written from the underlying transport, ignoring other restrictions that might otherwise keep them from being written. Its detailed function depends on the type of the bufferevent.

The iotype argument should be EV\_READ, EV\_WRITE, or EV\_READ|EV\_WRITE to indicate whether bytes being read, writ-ten, or both should be processed. The state argument may be one of BEV\_NORMAL, BEV\_FLUSH, or BEV\_FINISHED. BEV\_FINISHED indicates that the other side should be told that no more data will be sent; the distinction between BEV\_NORMAL and BEV\_FLUSH depends on the type of the bufferevent.

The bufferevent\_flush() function returns -1 on failure, 0 if no data was flushed, or 1 if some data was flushed.

Currently (as of Libevent 2.0.5-beta), bufferevent\_flush() is only implemented for some bufferevent types. In particular, socket-based bufferevents don’t have it.

**Type-specific bufferevent functions**

These bufferevent functions are not supported on all bufferevent types.

Interface

**int** bufferevent\_priority\_set(**struct** bufferevent\*bufev, **int** pri); **int** bufferevent\_get\_priority(**struct** bufferevent\*bufev);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 72 / 120 |

This function adjusts the priority of the events used to implement bufev to pri. See event\_priority\_set() for more information on priorities.

This function returns 0 on success, and -1 on failure. It works on socket-based bufferevents only.

The bufferevent\_priority\_set() function was introduced in Libevent 1.0; bufferevent\_get\_priority() didn’t appear until Libevent 2.1.2-alpha.

Interface

**int** bufferevent\_setfd(**struct** bufferevent\*bufev, evutil\_socket\_t fd);evutil\_socket\_t bufferevent\_getfd(**struct** bufferevent \*bufev);

These functions set or return the file descriptor for a fd-based event. Only socket-based bufferevents support setfd(). Both return -1 on failure; setfd() returns 0 on success.

The bufferevent\_setfd() function was introduced in Libevent 1.4.4; the bufferevent\_getfd() function was introduced in Libevent 2.0.2-alpha.

Interface

**struct** event\_base\*bufferevent\_get\_base(**struct** bufferevent\*bev);

This function returns the event\_base of a bufferevent. It was introduced in 2.0.9-rc.

Interface

**struct** bufferevent\*bufferevent\_get\_underlying(**struct** bufferevent\*bufev);

This function returns the bufferevent that another bufferevent is using as a transport, if any. For information on when this situation would occur, see notes on filtering bufferevents.

This function was introduced in Libevent 2.0.2-alpha.

**Manually locking and unlocking a bufferevent**

As with evbuffers, sometimes you want to ensure that a number of operations on a bufferevent are all performed atomically.

Libevent exposes functions that you can use to manually lock and unlock a bufferevent.

Interface

**void** bufferevent\_lock(**struct** bufferevent\*bufev); **void** bufferevent\_unlock(**struct** bufferevent\*bufev);

Note that locking a bufferevent has no effect if the bufferevent was not given the BEV\_OPT\_THREADSAFE thread on creation, or if Libevent’s threading support wasn’t activated.

Locking the bufferevent with this function will lock its associated evbuffers as well. These functions are recursive: it is safe to lock a bufferevent for which you already hold the lock. You must, of course, call unlock once for every time that you locked the bufferevent.

These functions were introduced in Libevent 2.0.6-rc.

**Obsolete bufferevent functionality**

The bufferevent backend code underwent substantial revision between Libevent 1.4 and Libevent 2.0. In the old interface, it was sometimes normal to build with access to the internals of the struct bufferevent, and to use macros that relied on this access.

To make matters confusing, the old code sometimes used names for bufferevent functionality that were prefixed with "evbuffer".

Here’s a brief guideline of what things used to be called before Libevent 2.0:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Fast portable non-blocking network programming with Libevent | | | 73 / 120 | |  |
|  |  |  |  |  |  |
|  | Current name |  | Old name |  |
|  | bufferevent\_data\_cb |  | evbuffercb |  |  |
|  | bufferevent\_event\_cb |  | everrorcb |  |  |
|  | BEV\_EVENT\_READING |  | EVBUFFER\_READ |  |  |
|  | BEV\_EVENT\_WRITE |  | EVBUFFER\_WRITE |  |  |
|  | BEV\_EVENT\_EOF |  | EVBUFFER\_EOF |  |  |
|  | BEV\_EVENT\_ERROR |  | EVBUFFER\_ERROR |  |  |
|  | BEV\_EVENT\_TIMEOUT |  | EVBUFFER\_TIMEOUT |  |  |
|  | bufferevent\_get\_input(b) |  | EVBUFFER\_INPUT(b) |  |  |
|  | bufferevent\_get\_output(b) |  | EVBUFFER\_OUTPUT(b) |  |  |

The old functions were defined in event.h, not in event2/bufferevent.h.

If you still need access to the internals of the common parts of the bufferevent struct, you can include event2/bufferevent\_struct.h. We recommend against it: the contents of struct bufferevent WILL change between versions of Libevent. The macros and names in this section are available if you include event2/bufferevent\_compat.h.

The interface to set up a bufferevent differed in older versions:

Interface

**struct** bufferevent\*bufferevent\_new(evutil\_socket\_t fd,

evbuffercb readcb, evbuffercb writecb, everrorcb errorcb, **void** \*cbarg); **int** bufferevent\_base\_set(**struct** event\_base\*base, **struct** bufferevent\*bufev);

The bufferevent\_new() function creates a socket bufferevent only, and does so on the deprecated "default" event\_base. Calling bufferevent\_base\_set adjusts the event\_base of a socket bufferevent only.

Instead of setting timeouts as struct timeval, they were set as numbers of seconds:

Interface

**void** bufferevent\_settimeout(**struct** bufferevent\*bufev,

**int** timeout\_read, **int** timeout\_write);

Finally, note that the underlying evbuffer implementation for Libevent versions before 2.0 was pretty inefficient, to the point where using bufferevents for high-performance apps was kind of questionable.

**Bufferevents: advanced topics**

This chapter describes some advanced features of Libevent’s bufferevent implementation that aren’t necessary for typical uses.

If you’re just learning how to use bufferevents, you should skip this chapter for now and go on to read [the evbuffer chapter](Ref7_evbuffer.html).

**Paired bufferevents**

Sometimes you have a networking program that needs to talk to itself. For example, you could have a program written to tunnel user connections over some protocol that sometimes also wants to tunnel connections of its own over that protocol. You could

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 74 / 120 |

achieve this by opening a connection to your own listening port and having your program use itself, of course, but that would waste resources by having your program talk to itself via the network stack.

Instead, you can create a pair of paired bufferevents such that all bytes written on one are received on the other (and vice versa), but no actual platform sockets are used.

Interface

**int** bufferevent\_pair\_new(**struct** event\_base\*base, **int** options,

**struct** bufferevent\*pair[2]);

Calling bufferevent\_pair\_new() sets pair[0] and pair[1] to a pair of bufferevents, each connected to the other. All the usual options are supported, except for BEV\_OPT\_CLOSE\_ON\_FREE, which has no effect, and BEV\_OPT\_DEFER\_CALLBACKS, which is always on.

Why do bufferevent pairs need to run with callbacks deferred? It’s pretty common for an operation on one element of the pair to invoke a callback that alters the bufferevent, thus invoking the other bufferevent’s callbacks, and so on through many steps. When the callbacks were not deferred, this chain of calls would pretty frequently overflow the stack, starve other connections, and require all the callbacks to be reentrant.

Paired bufferevents support flushing; setting the mode argument to either either BEV\_NORMAL or BEV\_FLUSH forces all the relevant data to get transferred from one bufferevent in the pair to the other, ignoring the watermarks that would otherwise restrict it. Setting mode to BEV\_FINISHED additionally generates an EOF event on the opposite bufferevent.

Freeing either member of the pair does not automatically free the other or generate an EOF event; it just makes the other member of the pair become unlinked. Once the bufferevent is unlinked, it will no longer successfully read or write data or generate any events.

Interface

**struct** bufferevent\*bufferevent\_pair\_get\_partner(**struct** bufferevent\*bev)

Sometimes you may need to get the other member of a bufferevent pair given only one member. To do this, you can invoke the bufferevent\_pair\_get\_partner() function. It will return the other member of the pair if bev is a member of a pair, and the other member still exists. Otherwise, it returns NULL.

Bufferevent pairs were new in Libevent 2.0.1-alpha; the bufferevent\_pair\_get\_partner() function was introduced in Libevent 2.0.6.

**Filtering bufferevents**

Sometimes you want to transform all the data passing through a bufferevent object. You could do this to add a compression layer, or wrap a protocol in another protocol for transport.

Interface

**enum** bufferevent\_filter\_result {

BEV\_OK = 0,

BEV\_NEED\_MORE = 1,

BEV\_ERROR = 2

};

**typedef enum** bufferevent\_filter\_result (\*bufferevent\_filter\_cb)(

**struct** evbuffer\*source, **struct** evbuffer\*destination, ev\_ssize\_t dst\_limit, **enum** bufferevent\_flush\_mode mode, **void** \*ctx);

**struct** bufferevent\*bufferevent\_filter\_new(**struct** bufferevent\*underlying,bufferevent\_filter\_cb input\_filter,

bufferevent\_filter\_cb output\_filter,

**int** options,

**void** (\*free\_context)(**void** \*),

**void** \*ctx);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 75 / 120 |

The bufferevent\_filter\_new() function creates a new filtering bufferevent, wrapped around an existing "underlying" bufferevent. All data received via the underlying bufferevent is transformed with the "input" filter before arriving at the filtering bufferevent, and all data sent via the filtering bufferevent is transformed with an "output" filter before being sent out to the underlying bufferevent.

Adding a filter to an underlying bufferevent replaces the callbacks on the underlying bufferevent. You can still add callbacks to the underlying bufferevent’s evbuffers, but you can’t set the callbacks on the bufferevent itself if you want the filter to still work.

The input\_filter and output\_filter functions are described below. All the usual options are supported in options. If BEV\_OPT\_CLOSE\_O is set, then freeing the filtering bufferevent also frees the underlying bufferevent. The ctx field is an arbitrary pointer passed to the filter functions; if a free\_context function is provided, it is called on ctx just before the filtering bufferevent is closed.

The input filter function will be called whenever there is new readable data on the underlying input buffer. The output filter function is called whenever there is new writable data on the filter’s output buffer. Each one receives a pair of evbuffers: a source evbuffer to read data from, and a destination evbuffer to write data to. The dst\_limit argument describes the upper bound of bytes to add to destination. The filter function is allowed to ignore this value, but doing so might violate high-water marks or rate limits. If dst\_limit is -1, there is no limit. The mode parameter tells the filter how aggressive to be in writing. If it is BEV\_NORMAL, then it should write as much as can be conveniently transformed. The BEV\_FLUSH value means to write as much as possible, and BEV\_FINISHED means that the filtering function should additionally do any cleanup necessary at the end of the stream. Finally, the filter function’s ctx argument is a void pointer as provided to the bufferevent\_filter\_new() constructor.

Filter functions must return BEV\_OK if any data was successfully written to the destination buffer, BEV\_NEED\_MORE if no more data can be written to the destination buffer without getting more input or using a different flush mode, and BEV\_ERROR if there is a non-recoverable error on the filter.

Creating the filter enables both reading and writing on the underlying bufferevent. You do not need to manage reads/writes on your own: the filter will suspend reading on the underlying bufferevent for you whenever it doesn’t want to read. For 2.0.8-rc and later, it is permissible to enable/disable reading and writing on the underlying bufferevent independently from the filter. If you do this, though, you may keep the filter from successfully getting the data it wants.

You don’t need to specify both an input filter and an output filter: any filter you omit is replaced with one that passes data on without transforming it.

**Limiting maximum single read/write size**

By default, bufferevents won’t read or write the maximum possible amount of bytes on each invocation of the event loop; doing so can lead to weird unfair behaviors and resource starvation. On the other hand, the defaults might not be reasonable for all situations.

Interface

**int** bufferevent\_set\_max\_single\_read(**struct** bufferevent\*bev, size\_t size); **int** bufferevent\_set\_max\_single\_write(**struct** bufferevent\*bev, size\_t size);

ev\_ssize\_t bufferevent\_get\_max\_single\_read(**struct** bufferevent \*bev); ev\_ssize\_t bufferevent\_get\_max\_single\_write(**struct** bufferevent \*bev);

The two "set" functions replace the current read and write maxima respectively. If the size value is 0 or above EV\_SSIZE\_MAX, they instead set the maxima to the default value. These functions return 0 on success and -1 on failure.

The two "get" functions return the current per-loop read and write maxima respectively.

These functions were added in 2.1.1-alpha.

**Bufferevents and Rate-limiting**

Some programs want to limit the amount of bandwidth used for any single bufferevent, or for a group of bufferevents. Libevent 2.0.4-alpha and Libevent 2.0.5-alpha added a basic facility to put caps on individual bufferevents, or to assign bufferevents to a rate-limited group.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 76 / 120 |

**The rate-limiting model**

Libevent’s rate-limiting uses a token bucket algorithm to decide how many bytes to read or write at a time. Every rate-limited object, at any given time, has a "read bucket" and a "write bucket", the sizes of which determine how many bytes the object is allowed to read or write immediately. Each bucket has a refill rate, a maximum burst size, and a timing unit or "tick". Whenever the timing unit elapses, the bucket is refilled proportionally to the refill rate—but if would become fuller than its burst size, any excess bytes are lost.

Thus, the refill rate determines the maximum average rate at which the object will send or receive bytes, and the burst size determines the largest number of bytes that will be sent or received in a single burst. The timing unit determines the smoothness of the traffic.

**Setting a rate limit on a bufferevent**

Interface

#define EV\_RATE\_LIMIT\_MAX EV\_SSIZE\_MAX

**struct** ev\_token\_bucket\_cfg;

**struct** ev\_token\_bucket\_cfg\*ev\_token\_bucket\_cfg\_new(

size\_t read\_rate, size\_t read\_burst,

size\_t write\_rate, size\_t write\_burst,

**const struct** timeval\*tick\_len);

**void** ev\_token\_bucket\_cfg\_free(**struct** ev\_token\_bucket\_cfg\*cfg); **int** bufferevent\_set\_rate\_limit(**struct** bufferevent\*bev,

**struct** ev\_token\_bucket\_cfg\*cfg);

An ev\_token\_bucket\_cfg structure represents the configuration values for a pair of token buckets used to limit reading and writing on a single bufferevent or group of bufferevents. To create one, call the ev\_token\_bucket\_cfg\_new function and provide the maximum average read rate, the maximum read burst, the maximum write rate, the maximum write burst, and the length of a tick. If the tick\_len argument is NULL, the length of a tick defaults to one second. The function may return NULL on error.

Note that the read\_rate and write\_rate arguments are scaled in units of bytes per tick. That is, a second, and read\_rate is 300, then the maximum average read rate is 3000 bytes per second. EV\_RATE\_LIMIT\_MAX are not supported.

if the tick is one tenth of Rate and burst values over

To limit a bufferevent’s transfer rate, call bufferevent\_set\_rate\_limit() on it with an ev\_token\_bucket\_cfg. The function returns 0 on success, and -1 on failure. You can give any number of bufferevents the same ev\_token\_bucket\_cfg. To remove a bufferevent’s rate limits, call bufferevent\_set\_rate\_limit(), passing NULL for the cfg parameter.

To free an ev\_token\_bucket\_cfg, call ev\_token\_bucket\_cfg\_free(). Note that it is NOT currently safe to do this until no buffer-events are using the ev\_token\_bucket\_cfg.

**Setting a rate limit on a group of bufferevents**

You can assign bufferevents to a rate limiting group if you want to limit their total bandwidth usage.

Interface

**struct** bufferevent\_rate\_limit\_group;

**struct** bufferevent\_rate\_limit\_group\*bufferevent\_rate\_limit\_group\_new( **struct** event\_base\*base,

**const struct** ev\_token\_bucket\_cfg\*cfg);

**int** bufferevent\_rate\_limit\_group\_set\_cfg(

**struct** bufferevent\_rate\_limit\_group\*group,

**const struct** ev\_token\_bucket\_cfg\*cfg);

**void** bufferevent\_rate\_limit\_group\_free(**struct** bufferevent\_rate\_limit\_group\*); **int** bufferevent\_add\_to\_rate\_limit\_group(**struct** bufferevent\*bev,

**struct** bufferevent\_rate\_limit\_group\*g);

**int** bufferevent\_remove\_from\_rate\_limit\_group(**struct** bufferevent\*bev);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 77 / 120 |

To construct a rate limiting group, call bufferevent\_rate\_limit\_group() with an event\_base and an initial ev\_token\_bucket\_cfg. You can add bufferevents to the group with bufferevent\_add\_to\_rate\_limit\_group() and bufferevent\_remove\_from\_rate\_limit\_group(); these functions return 0 on success and -1 on error.

A single bufferevent can be a member of no more than one rate limiting group at a time. A bufferevent can have both an individual rate limit (as set with bufferevent\_set\_rate\_limit()) and a group rate limit. When both limits are set, the lower limit for each bufferevent applies.

You can change the rate limit for an existing group by calling bufferevent\_rate\_limit\_group\_set\_cfg(). It returns 0 on success and -1 on failure. The bufferevent\_rate\_limit\_group\_free() function frees a rate limit group and removes all of its members.

As of version 2.0, Libevent’s group rate limiting tries to be fair on aggregate, but the implementation can be unfair on very small timescales. If you care strongly about scheduling fairness, please help out with patches for future versions.

**Inspecting current rate-limit values**

Sometimes your code may want to inspect the current rate limits that apply for a given bufferevent or group. Libevent provides some functions to do so.

Interface

ev\_ssize\_t bufferevent\_get\_read\_limit(**struct** bufferevent \*bev);

ev\_ssize\_t bufferevent\_get\_write\_limit(**struct** bufferevent \*bev);

ev\_ssize\_t bufferevent\_rate\_limit\_group\_get\_read\_limit(

**struct** bufferevent\_rate\_limit\_group\*);

ev\_ssize\_t bufferevent\_rate\_limit\_group\_get\_write\_limit(

**struct** bufferevent\_rate\_limit\_group\*);

The above functions return the current size, in bytes, of a bufferevent’s or a group’s read or write token buckets. Note that these values can be negative if a bufferevent has been forced to exceed its allocations. (Flushing the bufferevent can do this.)

Interface

ev\_ssize\_t bufferevent\_get\_max\_to\_read(**struct** bufferevent \*bev); ev\_ssize\_t bufferevent\_get\_max\_to\_write(**struct** bufferevent \*bev);

These functions return the number of bytes that a bufferevent would be willing to read or write right now, taking into account any rate limits that apply to the bufferevent, its rate limiting group (if any), and any maximum-to-read/write-at-a-time values imposed by Libevent as a whole.

Interface

**void** bufferevent\_rate\_limit\_group\_get\_totals(

**struct** bufferevent\_rate\_limit\_group\*grp,

ev\_uint64\_t \*total\_read\_out, ev\_uint64\_t \*total\_written\_out); **void** bufferevent\_rate\_limit\_group\_reset\_totals(

**struct** bufferevent\_rate\_limit\_group\*grp);

Each bufferevent\_rate\_limit\_group tracks the total number of bytes sent over it, in total. You can use this to track total usage by a number of bufferevents in the group. Calling bufferevent\_rate\_limit\_group\_get\_totals() on a group sets \*total\_read\_out and \*total\_written\_out to the total number of bytes read and written on a bufferevent group respectively. These totals start at 0 when the group is created, and reset to 0 whenever bufferevent\_rate\_limit\_group\_reset\_totals() is called on a group.

**Manually adjusting rate limits**

For programs with really complex needs, you might want to adjust the current values of a token bucket. You might want to do this, for example, if your program is generating traffic in some way that isn’t via a bufferevent.

Interface

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 78 / 120 |

**int** bufferevent\_decrement\_read\_limit(**struct** bufferevent\*bev, ev\_ssize\_t decr); **int** bufferevent\_decrement\_write\_limit(**struct** bufferevent\*bev, ev\_ssize\_t decr); **int** bufferevent\_rate\_limit\_group\_decrement\_read(

**struct** bufferevent\_rate\_limit\_group\*grp, ev\_ssize\_t decr); **int** bufferevent\_rate\_limit\_group\_decrement\_write(

**struct** bufferevent\_rate\_limit\_group\*grp, ev\_ssize\_t decr);

These functions decrement a current read or write bucket in a bufferevent or rate limiting group. Note that the decrements are signed: if you want to increment a bucket, pass a negative value.

**Setting the smallest share possible in a rate-limited group**

Frequently, you don’t want to divide the bytes available in a rate-limiting group up evenly among all bufferevents in every tick. For example, if you had 10,000 active bufferevents in a rate-limiting group with 10,000 bytes available for writing every tick, it wouldn’t be efficient to let each bufferevent write only 1 byte per tick, due to the overheads of system calls and TCP headers.

To solve this, each rate-limiting group has a notion of its "minimum share". In the situation above, instead of every bufferevent being allowed to write 1 byte per tick, 10,000/SHARE bufferevents will be allowed to write SHARE bytes each every tick, and the rest will be allowed to write nothing. Which bufferevents are allowed to write first is chosen randomly each tick.

The default minimum share is chosen to give decent performance, and is currently (as of 2.0.6-rc) set to 64. You can adjust this value with the following function:

Interface

**int** bufferevent\_rate\_limit\_group\_set\_min\_share(

**struct** bufferevent\_rate\_limit\_group\*group, size\_t min\_share);

Setting the min\_share to 0 disables the minimum-share code entirely.

Libevent’s rate-limiting has had minimum shares since it was first introduced. The function to change them was first exposed in Libevent 2.0.6-rc.

**Limitations of the rate-limiting implementation**

As of Libevent 2.0, there are some limitations to the rate-limiting implementation that you should know.

* Not every bufferevent type supports rate limiting well, or at all.
* Bufferevent rate limiting groups cannot nest, and a bufferevent can only be in a single rate limiting group at a time.
* The rate limiting implementation only counts bytes transferred in TCP packets as data, doesn’t include TCP headers.
* The read-limiting implementation relies on the TCP stack noticing that the application is only consuming data at a certain rate, and pushing back on the other side of the TCP connection when its buffers get full.
* Some implementations of bufferevents (particularly the windows IOCP implementation) can over-commit.
* Buckets start out with one full tick’s worth of traffic. This means that a bufferevent can start reading or writing immediately, and not wait until a full tick has passed. It also means, though, that a bufferevent that has been rate limited for N.1 ticks can potentially transfer N+1 ticks worth of traffic.
* Ticks cannot be smaller than 1 millisecond, and all fractions of a millisecond are ignored.

/// TODO: Write an example for rate-limiting

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 79 / 120 |

**Bufferevents and SSL**

Bufferevents can use the OpenSSL library to implement the SSL/TLS secure transport layer. Because many applications don’t need or want to link OpenSSL, this functionality is implemented in a separate library installed as "libevent\_openssl". Future versions of Libevent could add support for other SSL/TLS libraries such as NSS or GnuTLS, but right now OpenSSL is all that’s there.

OpenSSL functionality was introduced in Libevent 2.0.3-alpha, though it didn’t work so well before Libevent 2.0.5-beta or Libevent 2.0.6-rc.

This section is not a tutorial on OpenSSL, SSL/TLS, or cryptography in general.

These functions are all declared in the header "event2/bufferevent\_ssl.h".

**Setting up and using an OpenSSL-based bufferevent**

Interface

**enum** bufferevent\_ssl\_state {

BUFFEREVENT\_SSL\_OPEN = 0,

BUFFEREVENT\_SSL\_CONNECTING = 1,

BUFFEREVENT\_SSL\_ACCEPTING = 2

};

**struct** bufferevent\*

bufferevent\_openssl\_filter\_new(**struct** event\_base \*base, **struct** bufferevent\*underlying,

SSL \*ssl,

**enum** bufferevent\_ssl\_state state,

**int** options);

**struct** bufferevent\*

bufferevent\_openssl\_socket\_new(**struct** event\_base \*base, evutil\_socket\_t fd,

SSL \*ssl,

**enum** bufferevent\_ssl\_state state,

**int** options);

You can create two kinds of SSL bufferevents: a filter-based bufferevent that communicates over another underlying bufferevent, or a socket-based bufferevent that tells OpenSSL to communicate with the network directly over. In either case, you must provide an SSL object and a description of the SSL object’s state. The state should be BUFFEREVENT\_SSL\_CONNECTING if the SSL is currently performing negotiation as a client, BUFFEREVENT\_SSL\_ACCEPTING if the SSL is currently performing negotiation as a server, or BUFFEREVENT\_SSL\_OPEN if the SSL handshake is done.

The usual options are accepted; BEV\_OPT\_CLOSE\_ON\_FREE makes the SSL object and the underlying fd or bufferevent get closed when the openssl bufferevent itself is closed.

Once the handshake is complete, the new bufferevent’s event callback gets invoked with BEV\_EVENT\_CONNECTED in flags.

If you’re creating a socket-based bufferevent and the SSL object already has a socket set, you do not need to provide the socket yourself: just pass -1. You can also set the fd later with bufferevent\_setfd().

/// TODO: Remove this once bufferevent\_shutdown() API has been finished.

Note that when BEV\_OPT\_CLOSE\_ON\_FREE is set on a SSL bufferevent, a clean shutdown will not be performed on the SSL connection. This has two problems: first, the connection will seem to have been "broken" by the other side, rather than having been closed cleanly: the other party will not be able to tell whether you closed the connection, or whether it was broken by an attacker or third party. Second, OpenSSL will treat the session as "bad", and removed from the session cache. This can cause significant performance degradation on SSL applications under load.

Currently the only workaround is to do lazy SSL shutdowns manually. While this breaks the TLS RFC, it will make sure that sessions will stay in cache once closed. The following code implements this workaround.

Example

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 80 / 120 |

SSL \*ctx = bufferevent\_openssl\_get\_ssl(bev);

/\*

* SSL\_RECEIVED\_SHUTDOWN tells SSL\_shutdown to act as if we had already
* received a close notify from the other end. SSL\_shutdown will then
* send the final close notify in reply. The other end will receive the
* close notify and send theirs. By this time, we will have already
* closed the socket and the other end’s real close notify will never be
* received. In effect, both sides will think that they have completed a
* clean shutdown and keep their sessions valid. This strategy will fail
* if the socket is not ready for writing, in which case this hack will
* lead to an unclean shutdown and lost session on the other end.

\*/

SSL\_set\_shutdown(ctx, SSL\_RECEIVED\_SHUTDOWN);

SSL\_shutdown(ctx);

bufferevent\_free(bev);

Interface

SSL \*bufferevent\_openssl\_get\_ssl(**struct** bufferevent \*bev);

This function returns the SSL object used by an OpenSSL bufferevent, or NULL if bev is not an OpenSSL-based bufferevent.

Interface

**unsigned long** bufferevent\_get\_openssl\_error(**struct** bufferevent\*bev);

This function returns the first pending OpenSSL error for a given bufferevent’s operations, or 0 if there was no pending error.

The error format is as returned by ERR\_get\_error() in the openssl library.

Interface

**int** bufferevent\_ssl\_renegotiate(**struct** bufferevent\*bev);

Calling this function tells the SSL to renegotiate, and the bufferevent to invoke appropriate callbacks. This is an advanced topic; you should generally avoid it unless you really know what you’re doing, especially since many SSL versions have had known security issues related to renegotiation.

Interface

**int** bufferevent\_openssl\_get\_allow\_dirty\_shutdown(**struct** bufferevent\*bev);

**void** bufferevent\_openssl\_set\_allow\_dirty\_shutdown(**struct** bufferevent\*bev,

**int** allow\_dirty\_shutdown);

All good versions of the SSL protocol (that is, SSLv3 and all TLS versions) support an authenticated shutdown operation that enables the parties to distinguish an intentional close from an accidental or maliciously induced termination in the underling buffer. By default, we treat anything besides a proper shutdown as an error on the connection. If the allow\_dirty\_shutdown flag is set to 1, however, we treat a close in the connection as a BEV\_EVENT\_EOF.

The allow\_dirty\_shutdown functions were added in Libevent 2.1.1-alpha.

Example: A simple SSL-based echo server

/\* Simple echo server using OpenSSL bufferevents \*/ #include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <openssl/ssl.h>

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 81 / 120 |

#include <openssl/err.h>

#include <openssl/rand.h>

#include <event.h>

#include <event2/listener.h>

#include <event2/bufferevent\_ssl.h>

**static void**

ssl\_readcb(**struct** bufferevent \* bev, **void** \* arg)

{

**struct** evbuffer\*in = bufferevent\_get\_input(bev);

printf("Received %zu bytes\n", evbuffer\_get\_length(in)); printf("----- data ----\n");

printf("%.\*s\n", (**int**)evbuffer\_get\_length(in), evbuffer\_pullup(in, -1));

bufferevent\_write\_buffer(bev, in);

}

**static void**

ssl\_acceptcb(**struct** evconnlistener \*serv, **int** sock, **struct** sockaddr \*sa, **int** sa\_len, **void** \*arg)

{

**struct** event\_base\*evbase;

**struct** bufferevent\*bev;

SSL\_CTX \*server\_ctx;

SSL \*client\_ctx;

server\_ctx = (SSL\_CTX \*)arg;

client\_ctx = SSL\_new(server\_ctx);

evbase = evconnlistener\_get\_base(serv);

bev = bufferevent\_openssl\_socket\_new(evbase, sock, client\_ctx,

BUFFEREVENT\_SSL\_ACCEPTING,

BEV\_OPT\_CLOSE\_ON\_FREE);

bufferevent\_enable(bev, EV\_READ);

bufferevent\_setcb(bev, ssl\_readcb, NULL, NULL, NULL);

}

**static** SSL\_CTX\*

evssl\_init(**void**)

{

SSL\_CTX \*server\_ctx;

/\* Initialize the OpenSSL library \*/

SSL\_load\_error\_strings();

SSL\_library\_init();

/\* We MUST have entropy, or else there’s no point to crypto. \*/ **if** (!RAND\_poll())

**return** NULL;

server\_ctx = SSL\_CTX\_new(SSLv23\_server\_method());

**if** (! SSL\_CTX\_use\_certificate\_chain\_file(server\_ctx, "cert") ||

* SSL\_CTX\_use\_PrivateKey\_file(server\_ctx, "pkey", SSL\_FILETYPE\_PEM)) {

puts("Couldn’t read ’pkey’ or ’cert’ file. To generate a key\n" "and self-**signed** certificate, run:\n"

* + openssl genrsa -out pkey 2048\n"
  + openssl req -**new** -key pkey -out cert.req\n"
  + openssl x509 -req -days 365 -in cert.req -signkey pkey -out cert"); **return** NULL;

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 82 / 120 |

}

SSL\_CTX\_set\_options(server\_ctx, SSL\_OP\_NO\_SSLv2);

**return** server\_ctx;

}

**int**

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

{

SSL\_CTX \*ctx;

**struct** evconnlistener\*listener;

**struct** event\_base\*evbase;

**struct** sockaddr\_in sin;

memset(&sin, 0, **sizeof**(sin));

sin.sin\_family = AF\_INET;

sin.sin\_port = htons(9999);

sin.sin\_addr.s\_addr = htonl(0x7f000001); /\* 127.0.0.1 \*/

ctx = evssl\_init();

**if** (ctx == NULL)

**return** 1;

evbase = event\_base\_new();

listener = evconnlistener\_new\_bind(

evbase, ssl\_acceptcb, (**void** \*)ctx,

LEV\_OPT\_CLOSE\_ON\_FREE | LEV\_OPT\_REUSEABLE, 1024, (**struct** sockaddr \*)&sin, **sizeof**(sin));

event\_base\_loop(evbase, 0);

evconnlistener\_free(listener);

SSL\_CTX\_free(ctx);

**return** 0;

}

**Some notes on threading and OpenSSL**

The built in threading mechanisms of Libevent do not cover OpenSSL locking. Since OpenSSL uses a myriad of global variables, you must still configure OpenSSL to be thread safe. While this process is outside the scope of Libevent, this topic comes up enough to warrant discussion.

Example: A very simple example of how to enable thread safe OpenSSL

/\*

* Please refer to OpenSSL documentation to verify you are doing this correctly,
* Libevent does not guarantee this code is the complete picture, but to be used
* only as an example.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#include <openssl/ssl.h>

#include <openssl/crypto.h>

pthread\_mutex\_t \* ssl\_locks;

**int** ssl\_num\_locks;

/\* Implements a thread-ID function as requied by openssl \*/

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 83 / 120 |

**static unsigned long**

get\_thread\_id\_cb(**void**)

{

**return** (**unsigned long**)pthread\_self();

}

**static void**

thread\_lock\_cb(**int** mode, **int** which, **const char** \* f, **int** l)

{

**if** (which < ssl\_num\_locks) {

**if** (mode & CRYPTO\_LOCK) {

pthread\_mutex\_lock(&(ssl\_locks[which]));

} **else** {

pthread\_mutex\_unlock(&(ssl\_locks[which]));

}

}

}

**int**

init\_ssl\_locking(**void**)

{

**int** i;

ssl\_num\_locks = CRYPTO\_num\_locks();

ssl\_locks = malloc(ssl\_num\_locks \* **sizeof**(pthread\_mutex\_t)); **if** (ssl\_locks == NULL)

**return** -1;

**for** (i = 0; i < ssl\_num\_locks; i++) {

pthread\_mutex\_init(&(ssl\_locks[i]), NULL);

}

CRYPTO\_set\_id\_callback(get\_thread\_id\_cb);

CRYPTO\_set\_locking\_callback(thread\_lock\_cb);

**return** 0;

}

**Evbuffers: utility functionality for buffered IO**

Libevent’s evbuffer functionality implements a queue of bytes, optimized for adding data to the end and removing it from the front.

Evbuffers are meant to be generally useful for doing the "buffer" part of buffered network IO. They do not provide functions to schedule the IO or trigger the IO when it’s ready: that is what bufferevents do.

The functions in this chapter are declared in event2/buffer.h unless otherwise noted.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 84 / 120 |

**Creating or freeing an evbuffer**

Interface

**struct** evbuffer\*evbuffer\_new(**void**);

**void** evbuffer\_free(**struct** evbuffer\*buf);

These functions should be relatively clear: evbuffer\_new() allocates and returns a new empty evbuffer, and evbuffer\_free() deletes one and all of its contents.

These functions have existed since Libevent 0.8.

**Evbuffers and Thread-safety**

Interface

**int** evbuffer\_enable\_locking(**struct** evbuffer\*buf, **void** \*lock);

**void** evbuffer\_lock(**struct** evbuffer\*buf);

**void** evbuffer\_unlock(**struct** evbuffer\*buf);

By default, it is not safe to access an evbuffer from multiple threads at once. If you need to do this, you can call evbuffer\_enable\_locking() on the evbuffer. If its lock argument is NULL, Libevent allocates a new lock using the lock creation function that was provided to evthread\_set\_lock\_creation\_callback. Otherwise, it uses the argument as the lock.

The evbuffer\_lock() and evbuffer\_unlock() functions acquire and release the lock on an evbuffer respectively. You can use them to make a set of operations atomic. If locking has not been enabled on the evbuffer, these functions do nothing.

(Note that you do not need to call evbuffer\_lock() and evbuffer\_unlock() around individual operations: if locking is enabled on the evbuffer, individual operations are already atomic. You only need to lock the evbuffer manually when you have more than one operation that need to execute without another thread butting in.)

These functions were all introduced in Libevent 2.0.1-alpha.

**Inspecting an evbuffer**

Interface

size\_t evbuffer\_get\_length(**const struct** evbuffer \*buf);

This function returns the number of bytes stored in an evbuffer.

It was introduced in Libevent 2.0.1-alpha.

Interface

size\_t evbuffer\_get\_contiguous\_space(**const struct** evbuffer \*buf);

This function returns the number of bytes stored contiguously at the front of the evbuffer. The bytes in an evbuffer may be stored in multiple separate chunks of memory; this function returns the number of bytes currently stored in the first chunk.

It was introduced in Libevent 2.0.1-alpha.

**Adding data to an evbuffer: basics**

Interface

**int** evbuffer\_add(**struct** evbuffer\*buf, **const void** \*data, size\_t datlen);

This function appends the datlen bytes in data to the end of buf. It returns 0 on success, and -1 on failure.

Interface

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 85 / 120 |

**int** evbuffer\_add\_printf(**struct** evbuffer\*buf, **const char** \*fmt, ...)

**int** evbuffer\_add\_vprintf(**struct** evbuffer\*buf, **const char** \*fmt, va\_list ap);

These functions append formatted data to the end of buf. The format argument and other remaining arguments are handled as if by the C library functions "printf" and "vprintf" respectively. The functions return the number of bytes appended.

Interface

**int** evbuffer\_expand(**struct** evbuffer\*buf, size\_t datlen);

This function alters the last chunk of memory in the buffer, or adds a new chunk, such that the buffer is now large enough to contain datlen bytes without any further allocations.

Examples

/\* Here are two ways to add "Hello world 2.0.1" to a buffer. \*/ /\* Directly: \*/

evbuffer\_add(buf, "Hello world 2.0.1", 17);

/\* Via printf: \*/

evbuffer\_add\_printf(buf, "Hello %s %d.%d.%d", "world", 2, 0, 1);

The evbuffer\_add() and evbuffer\_add\_printf() functions were introduced in Libevent 0.8; evbuffer\_expand() was in Libevent 0.9, and evbuffer\_add\_vprintf() first appeared in Libevent 1.1.

**Moving data from one evbuffer to another**

For efficiency, Libevent has optimized functions for moving data from one evbuffer to another.

Interface

**int** evbuffer\_add\_buffer(**struct** evbuffer\*dst, **struct** evbuffer\*src); **int** evbuffer\_remove\_buffer(**struct** evbuffer\*src, **struct** evbuffer\*dst,

size\_t datlen);

The evbuffer\_add\_buffer() function moves all data from src to the end of dst. It returns 0 on success, -1 on failure.

The evbuffer\_remove\_buffer() function moves exactly datlen bytes from src to the end of dst, copying as little as possible. If there are fewer than datlen bytes to move, it moves all the bytes. It returns the number of bytes moved.

We introduced evbuffer\_add\_buffer() in Libevent 0.8; evbuffer\_remove\_buffer() was new in Libevent 2.0.1-alpha.

**Adding data to the front of an evbuffer**

Interface

**int** evbuffer\_prepend(**struct** evbuffer\*buf, **const void** \*data, size\_t size); **int** evbuffer\_prepend\_buffer(**struct** evbuffer\*dst, **struct** evbuffer\*src);

These functions behave as evbuffer\_add() and evbuffer\_add\_buffer() respectively, except that they move data to the front of the destination buffer.

These functions should be used with caution, and never on an evbuffer shared with a bufferevent. They were new in Libevent 2.0.1-alpha.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 86 / 120 |

**Rearranging the internal layout of an evbuffer**

Sometimes you want to peek at the first N bytes of data in the front of an evbuffer, and see it as a contiguous array of bytes. To do this, you must first ensure that the front of the buffer really is contiguous.

Interface

**unsigned char** \*evbuffer\_pullup(**struct** evbuffer\*buf, ev\_ssize\_t size);

The evbuffer\_pullup() function "linearizes" the first size bytes of buf, copying or moving them as needed to ensure that they are all contiguous and occupying the same chunk of memory. If size is negative, the function linearizes the entire buffer. If size is greater than the number of bytes in the buffer, the function returns NULL. Otherwise, evbuffer\_pullup() returns a pointer to the first byte in buf.

Calling evbuffer\_pullup() with a large size can be quite slow, since it potentially needs to copy the entire buffer’s contents.

Example

#include <event2/buffer.h>

#include <event2/util.h>

#include <string.h>

**int** parse\_socks4(**struct** evbuffer\*buf, ev\_uint16\_t\*port, ev\_uint32\_t\*addr){

/\* Let’s parse the start of a SOCKS4 request! The format is easy:

* 1 byte of version, 1 byte of command, 2 bytes destport, 4 bytes of
* destip. \*/

**unsigned char** \*mem;

mem = evbuffer\_pullup(buf, 8);

**if** (mem == NULL) {

/\* Not enough data in the buffer \*/

**return** 0;

} **else if** (mem[0] != 4 || mem[1] != 1) {

/\* Unrecognized protocol or command \*/

**return** -1;

} **else** {

memcpy(port, mem+2, 2);

memcpy(addr, mem+4, 4);

\*port = ntohs(\*port);

\*addr = ntohl(\*addr);

/\* Actually remove the data from the buffer now that we know we like it. \*/

evbuffer\_drain(buf, 8);

**return** 1;

}

}

Note Calling evbuffer\_pullup() with size equal to the value returned by evbuffer\_get\_contiguous\_space() will not result in any data being copied or moved.

The evbuffer\_pullup() function was new in Libevent 2.0.1-alpha: previous versions of Libevent always kept evbuffer data con-tiguous, regardless of the cost.

**Removing data from an evbuffer**

Interface

**int** evbuffer\_drain(**struct** evbuffer\*buf, size\_t len);

**int** evbuffer\_remove(**struct** evbuffer\*buf, **void** \*data, size\_t datlen);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 87 / 120 |

The evbuffer\_remove() function copies and removes the first datlen bytes from the front of buf into the memory at data. If there are fewer than datlen bytes available, the function copies all the bytes there are. The return value is -1 on failure, and is otherwise the number of bytes copied.

The evbuffer\_drain() function behaves as evbuffer\_remove(), except that it does not copy the data: it just removes it from the front of the buffer. It returns 0 on success and -1 on failure.

Libevent 0.8 introduced evbuffer\_drain(); evbuffer\_remove() appeared in Libevent 0.9.

**Copying data out from an evbuffer**

Sometimes you want to get a copy of the data at the start of a buffer without draining it. For example, you might want to see whether a complete record of some kind has arrived, without draining any of the data (as evbuffer\_remove would do), or rearranging the buffer internally (as evbuffer\_pullup() would do.)

Interface

ev\_ssize\_t evbuffer\_copyout(**struct** evbuffer \*buf, **void** \*data, size\_t datlen); ev\_ssize\_t evbuffer\_copyout\_from(**struct** evbuffer \*buf,

**const struct** evbuffer\_ptr\*pos,

**void** \*data\_out, size\_t datlen);

The evbuffer\_copyout() behaves just like evbuffer\_remove(), but does not drain any data from the buffer. That is, it copies the first datlen bytes from the front of buf into the memory at data. If there are fewer than datlen bytes available, the function copies all the bytes there are. The return value is -1 on failure, and is otherwise the number of bytes copied.

The evbuffer\_copyout\_from() function behaves like evbuffer\_copyout(), but instead of copying bytes from the front of the buffer, it copies them beginning at the position provided in pos. See "Searching within an evbuffer" below for information on the evbuffer\_ptr structure.

If copying data from the buffer is too slow, use evbuffer\_peek() instead.

Example

#include <event2/buffer.h>

#include <event2/util.h>

#include <stdlib.h>

#include <stdlib.h>

**int** get\_record(**struct** evbuffer\*buf, size\_t\*size\_out, **char** \*\*record\_out)

{

/\* Let’s assume that we’re speaking some protocol where records contain a 4-byte size field in network order, followed by that number of bytes. We will return 1 and set the ’out’ fields if we have a whole record, return 0 if the record isn’t here yet, and -1 on error. \*/

size\_t buffer\_len = evbuffer\_get\_length(buf);

ev\_uint32\_t record\_len;

**char** \*record;

**if** (buffer\_len < 4)

**return** 0; /\*The size field hasn’t arrived.\*/

/\* We use evbuffer\_copyout here so that the size field will stay on the buffer for now. \*/

evbuffer\_copyout(buf, &record\_len, 4);

/\* Convert len\_buf into host order. \*/

record\_len = ntohl(record\_len);

**if** (buffer\_len < record\_len + 4)

**return** 0; /\*The record hasn’t arrived\*/

/\* Okay, \_now\_ we can remove the record. \*/

record = malloc(record\_len);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 88 / 120 |

**if** (record == NULL)

**return** -1;

evbuffer\_drain(buf, 4);

evbuffer\_remove(buf, record, record\_len);

\*record\_out = record;

\*size\_out = record\_len;

**return** 1;

}

The evbuffer\_copyout() function first appeared in Libevent 2.0.5-alpha; evbuffer\_copyout\_from() was added in Libevent 2.1.1-alpha.

**Line-oriented input**

Interface

**enum** evbuffer\_eol\_style {

EVBUFFER\_EOL\_ANY,

EVBUFFER\_EOL\_CRLF,

EVBUFFER\_EOL\_CRLF\_STRICT,

EVBUFFER\_EOL\_LF,

EVBUFFER\_EOL\_NUL

};

**char** \*evbuffer\_readln(**struct** evbuffer\*buffer, size\_t\*n\_read\_out, **enum** evbuffer\_eol\_style eol\_style);

Many Internet protocols use line-based formats. The evbuffer\_readln() function extracts a line from the front of an evbuffer and returns it in a newly allocated NUL-terminated string. If n\_read\_out is not NULL, \*n\_read\_out is set to the number of bytes in the string returned. If there is not a whole line to read, the function returns NULL. The line terminator is not included in the copied string.

The evbuffer\_readln() function understands 4 line termination formats:

EVBUFFER\_EOL\_LF

The end of a line is a single linefeed character. (This is also known as "\n". It is ASCII value is 0x0A.)

EVBUFFER\_EOL\_CRLF\_STRICT

The end of a line is a single carriage return, followed by a single linefeed. (This is also known as "\r\n". The ASCII values are 0x0D 0x0A).

EVBUFFER\_EOL\_CRLF

The end of the line is an optional carriage return, followed by a linefeed. (In other words, it is either a "\r\n" or a "\n".) This format is useful in parsing text-based Internet protocols, since the standards generally prescribe a "\r\n" line-terminator, but nonconformant clients sometimes say just "\n".

EVBUFFER\_EOL\_ANY

The end of line is any sequence of any number of carriage return and linefeed characters. This format is not very useful; it exists mainly for backward compatibility.

EVBUFFER\_EOL\_NUL

The end of line is a single byte with the value 0 — that is, an ASCII NUL.

(Note that if you used event\_set\_mem\_functions() to override the default malloc, the string returned by evbuffer\_readln will be allocated by the malloc-replacement you specified.)

Example

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 89 / 120 |

**char** \*request\_line;

size\_t len;

request\_line = evbuffer\_readln(buf, &len, EVBUFFER\_EOL\_CRLF); **if** (!request\_line) {

/\* The first line has not arrived yet. \*/

} **else** {

**if** (!strncmp(request\_line, "HTTP/1.0 ", 9)) {

/\* HTTP 1.0 detected ... \*/

}

free(request\_line);

}

The evbuffer\_readln() interface is available in Libevent 1.4.14-stable and later. EVBUFFER\_EOL\_NUL was added in Libevent 2.1.1-alpha.

**Searching within an evbuffer**

The evbuffer\_ptr structure points to a location within an evbuffer, and contains data that you can use to iterate through an evbuffer.

Interface

**struct** evbuffer\_ptr {

ev\_ssize\_t pos;

**struct** {

/\* internal fields \*/

} \_internal;

};

The pos field is the only public field; the others should not be used by user code. It indicates a position in the evbuffer as an offset from the start.

Interface

**struct** evbuffer\_ptr evbuffer\_search(**struct** evbuffer\*buffer,

**const char** \*what, size\_t len, **const struct** evbuffer\_ptr\*start); **struct** evbuffer\_ptr evbuffer\_search\_range(**struct** evbuffer\*buffer,

**const char** \*what, size\_t len, **const struct** evbuffer\_ptr\*start, **const struct** evbuffer\_ptr\*end);

**struct** evbuffer\_ptr evbuffer\_search\_eol(**struct** evbuffer\*buffer, **struct** evbuffer\_ptr\*start, size\_t\*eol\_len\_out,

**enum** evbuffer\_eol\_style eol\_style);

The evbuffer\_search() function scans the buffer for an occurrence of the len-character string what. It returns an evbuffer\_ptr containing the position of the string, or -1 if the string was not found. If the start argument is provided, it’s the position at which the search should begin; otherwise, the search is from the start of the string.

The evbuffer\_search\_range() function behaves as evbuffer\_search, except that it only considers occurrences of what that occur before the evbuffer\_ptr end.

The evbuffer\_search\_eol() function detects line-endings as evbuffer\_readln(), but instead of copying out the line, returns an evbuffer\_ptr to the start of the end-of-line characters(s). If eol\_len\_out is non-NULL, it is set to the length of the EOL string.

Interface

**enum** evbuffer\_ptr\_how {

EVBUFFER\_PTR\_SET,

EVBUFFER\_PTR\_ADD

};

**int** evbuffer\_ptr\_set(**struct** evbuffer \*buffer, **struct** evbuffer\_ptr \*pos, size\_t position, **enum** evbuffer\_ptr\_how how);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 90 / 120 |

The evbuffer\_ptr\_set function manipulates the position of an evbuffer\_ptr pos within buffer. If how is EVBUFFER\_PTR\_SET, the pointer is moved to an absolute position position within the buffer. If it is EVBUFFER\_PTR\_ADD, the pointer moves position bytes forward. This function returns 0 on success and -1 on failure.

Example

#include <event2/buffer.h>

#include <string.h>

/\* Count the total occurrences of ’str’ in ’buf’. \*/

**int** count\_instances(**struct** evbuffer\*buf, **const char** \*str)

{

size\_t len = strlen(str);

**int** total = 0;

**struct** evbuffer\_ptr p;

**if** (!len)

/\* Don’t try to count the occurrences of a 0-length string. \*/ **return** -1;

evbuffer\_ptr\_set(buf, &p, 0, EVBUFFER\_PTR\_SET);

**while** (1) {

p = evbuffer\_search(buf, str, len, &p);

**if** (p.pos < 0)

**break**;

total++;

evbuffer\_ptr\_set(buf, &p, 1, EVBUFFER\_PTR\_ADD);

}

**return** total;

}

WARNING Any call that modifies an evbuffer or its layout invalidates all outstanding evbuffer\_ptr values, and makes them unsafe to use.

These interfaces were new in Libevent 2.0.1-alpha.

**Inspecting data without copying it**

Sometimes, you want to read data in an evbuffer without copying it out (as evbuffer\_copyout() does), and without rearranging the evbuffer’s internal memory (as evbuffer\_pullup() does). Sometimes you might want to see data in the middle of an evbuffer.

You can do this with:

Interface

**struct** evbuffer\_iovec {

**void** \*iov\_base;

size\_t iov\_len;

};

**int** evbuffer\_peek(**struct** evbuffer \*buffer, ev\_ssize\_t len, **struct** evbuffer\_ptr\*start\_at,

**struct** evbuffer\_iovec\*vec\_out, **int** n\_vec);

When you call evbuffer\_peek(), you give it an array of evbuffer\_iovec structures in vec\_out. The array’s length is n\_vec. It sets these structures so that each one contains a pointer to a chunk of the evbuffer’s internal RAM (iov\_base), and the length of memory that is set in that chunk.

If len is less than 0, evbuffer\_peek() tries to fill all of the evbuffer\_iovec structs you have given it. Otherwise, it fills them until either they are all used, or at least len bytes are visible. If the function could give you all the data you asked for, it returns the

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 91 / 120 |

number of evbuffer\_iovec structures that it actually used. Otherwise, it returns the number that it would need in order to give what you asked for.

When ptr is NULL, evbuffer\_peek() starts at the beginning of the buffer. Otherwise, it starts at the pointer given in ptr. Examples

{

/\* Let’s look at the first two chunks of buf, and write them to stderr. \*/ **int** n, i;

**struct** evbuffer\_iovec v[2];

n = evbuffer\_peek(buf, -1, NULL, v, 2);

**for** (i=0; i<n; ++i) { /\* There might be less than two chunks available. \*/ fwrite(v[i].iov\_base, 1, v[i].iov\_len, stderr);

}

}

{

/\* Let’s send the first 4906 bytes to stdout via write. \*/ **int** n, i, r;

**struct** evbuffer\_iovec\*v;

size\_t written = 0;

/\* determine how many chunks we need. \*/

n = evbuffer\_peek(buf, 4096, NULL, NULL, 0);

/\* Allocate space for the chunks. This would be a good time to use alloca() if you have it. \*/

v = malloc(**sizeof**(**struct** evbuffer\_iovec)\*n);

/\* Actually fill up v. \*/

n = evbuffer\_peek(buf, 4096, NULL, v, n);

**for** (i=0; i<n; ++i) {

size\_t len = v[i].iov\_len;

**if** (written + len > 4096)

len = 4096 - written;

r = write(1 /\* stdout \*/, v[i].iov\_base, len);

**if** (r<=0)

**break**;

/\* We keep track of the bytes written separately; if we don’t,

we may write more than 4096 bytes if the last chunk puts us over the limit. \*/

written += len;

}

free(v);

}

{

/\* Let’s get the first 16K of data after the first occurrence of the string "start\n", and pass it to a consume() function. \*/

**struct** evbuffer\_ptr ptr;

**struct** evbuffer\_iovec v[1];

**const char** s[] = "start\n";

**int** n\_written;

ptr = evbuffer\_search(buf, s, strlen(s), NULL);

**if** (ptr.pos == -1)

**return**; /\*no start string found.\*/

/\* Advance the pointer past the start string. \*/

**if** (evbuffer\_ptr\_set(buf, &ptr, strlen(s), EVBUFFER\_PTR\_ADD) < 0)

**return**; /\*off the end of the string.\*/

**while** (n\_written < 16\*1024) {

/\* Peek at a single chunk. \*/

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 92 / 120 |

**if** (evbuffer\_peek(buf, -1, &ptr, v, 1) < 1)

**break**;

/\* Pass the data to some user-defined consume function \*/ consume(v[0].iov\_base, v[0].iov\_len);

n\_written += v[0].iov\_len;

/\* Advance the pointer so we see the next chunk next time. \*/

**if** (evbuffer\_ptr\_set(buf, &ptr, v[0].iov\_len, EVBUFFER\_PTR\_ADD)<0)

**break**;

}

}

NOTES

* Modifying the data pointed to by the evbuffer\_iovec can result in undefined behavior.
* If any function is called that modifies the evbuffer, the pointers that evbuffer\_peek() yields may become invalid.
* If your evbuffer could be used in multiple threads, make sure to lock it with evbuffer\_lock() before you call evbuffer\_peek(), and unlock it once you are done using the extents that evbuffer\_peek() gave you.

This function is new in Libevent 2.0.2-alpha.

**Adding data to an evbuffer directly**

Sometimes you want to insert data info an evbuffer directly, without first writing it into a character array and then copying it in with evbuffer\_add(). There are an advanced pair of functions you can use to do this: evbuffer\_reserve\_space() and evbuffer\_commit\_space(). As with evbuffer\_peek(), these functions use the evbuffer\_iovec structure to provide direct access to memory inside the evbuffer.

Interface

**int** evbuffer\_reserve\_space(**struct** evbuffer \*buf, ev\_ssize\_t size, **struct** evbuffer\_iovec\*vec, **int** n\_vecs);

**int** evbuffer\_commit\_space(**struct** evbuffer\*buf,

**struct** evbuffer\_iovec\*vec, **int** n\_vecs);

The evbuffer\_reserve\_space() function gives you pointers to space inside the evbuffer. It expands the buffer as necessary to give you at least size bytes. The pointers to these extents, and their lengths, will be stored in the array of vectors you pass in with vec; n\_vec is the length of this array.

The value of n\_vec must be at least 1. If you provide only one vector, then Libevent will ensure that you have all the contiguous space you requested in a single extent, but it may have to rearrange the buffer or waste memory in order to do so. For better performance, provide at least 2 vectors. The function returns the number of provided vectors that it needed for the space you requested.

The data that you write into these vectors is not part of the buffer until you call evbuffer\_commit\_space(), which actually makes the data you wrote count as being in the buffer. If you want to commit less space than you asked for, you can decrease the iov\_len field in any of the evbuffer\_iovec structures you were given. You can also pass back fewer vectors than you were given. The evbuffer\_commit\_space() function returns 0 on success and -1 on failure.

NOTES AND CAVEATS

* Calling any function that rearranges the evbuffer or adds data to it evbuffer will invalidate the pointers you got from evbuffer\_reserve\_sp
* In the current implementation, evbuffer\_reserve\_space() never uses more than two vectors, no matter how many the user supplies. This may change in a future release.
* It is safe to call evbuffer\_reserve\_space() any number of times.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 93 / 120 |

* If your evbuffer could be used in multiple threads, make sure to lock it with evbuffer\_lock() before you call evbuffer\_reserve\_space(), and unlock it once you commit.

Example

/\* Suppose we want to fill a buffer with 2048 bytes of output from a generate\_data() function, without copying. \*/

**struct** evbuffer\_iovec v[2];

**int** n, i;

size\_t n\_to\_add = 2048;

/\* Reserve 2048 bytes.\*/

n = evbuffer\_reserve\_space(buf, n\_to\_add, v, 2);

**if** (n<=0)

**return**; /\*Unable to reserve the space **for** some reason.\*/

**for** (i=0; i<n && n\_to\_add > 0; ++i) {

size\_t len = v[i].iov\_len;

**if** (len > n\_to\_add) /\* Don’t write more than n\_to\_add bytes. \*/ len = n\_to\_add;

**if** (generate\_data(v[i].iov\_base, len) < 0) {

/\* If there was a problem during data generation, we can just stop here; no data will be committed to the buffer. \*/

**return**;

}

/\* Set iov\_len to the number of bytes we actually wrote, so we don’t commit too much. \*/

v[i].iov\_len = len;

}

/\* We commit the space here. Note that we give it ’i’ (the number of vectors we actually used) rather than ’n’ (the number of vectors we had available. \*/

**if** (evbuffer\_commit\_space(buf, v, i) < 0)

**return**; /\*Error committing\*/

Bad Examples

/\* Here are some mistakes you can make with evbuffer\_reserve().

DO NOT IMITATE THIS CODE. \*/

**struct** evbuffer\_iovec v[2];

{

/\* Do not use the pointers from evbuffer\_reserve\_space() after calling any functions that modify the buffer. \*/

evbuffer\_reserve\_space(buf, 1024, v, 2);

evbuffer\_add(buf, "X", 1);

/\* WRONG: This next line won’t work if evbuffer\_add needed to rearrange the buffer’s contents. It might even crash your program. Instead, you add the data before calling evbuffer\_reserve\_space. \*/

memset(v[0].iov\_base, ’Y’, v[0].iov\_len-1);

evbuffer\_commit\_space(buf, v, 1);

}

{

/\* Do not modify the iov\_base pointers. \*/

**const char** \*data = "Here is some data";

evbuffer\_reserve\_space(buf, strlen(data), v, 1);

/\* WRONG: The next line will not do what you want. Instead, you should \_copy\_ the contents of data into v[0].iov\_base. \*/

v[0].iov\_base = (**char**\*) data;

v[0].iov\_len = strlen(data);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 94 / 120 |

/\* In this case, evbuffer\_commit\_space might give an error if you’re

lucky \*/

evbuffer\_commit\_space(buf, v, 1);

}

These functions have existed with their present interfaces since Libevent 2.0.2-alpha.

**Network IO with evbuffers**

The most common use case for evbuffers in Libevent is network IO. The interface for performing network IO on an evbuffer is:

Interface

**int** evbuffer\_write(**struct** evbuffer\*buffer, evutil\_socket\_t fd);

**int** evbuffer\_write\_atmost(**struct** evbuffer\*buffer, evutil\_socket\_t fd,ev\_ssize\_t howmuch);

**int** evbuffer\_read(**struct** evbuffer\*buffer, evutil\_socket\_t fd, **int** howmuch);

The evbuffer\_read() function reads up to howmuch bytes from the socket fd onto the end of buffer. It returns a number of bytes read on success, 0 on EOF, and -1 on an error. Note that the error may indicate that a nonblocking operation would not succeed; you need to check the error code for EAGAIN (or WSAEWOULDBLOCK on Windows). If howmuch is negative, evbuffer\_read() tries to guess how much to read itself.

The evbuffer\_write\_atmost() function tries to write up to howmuch bytes from the front of buffer onto the socket fd. It returns a number of bytes written on success, and -1 on failure. As with evbuffer\_read(), you need to check the error code to see whether the error is real, or just indicates that nonblocking IO could not be completed immediately. If you give a negative value for howmuch, we try to write the entire contents of the buffer.

Calling evbuffer\_write() is the same as calling evbuffer\_write\_atmost() with a negative howmuch argument: it attempts to flush as much of the buffer as it can.

On Unix, these functions should work on any file descriptor that supports read and write. On Windows, only sockets are supported.

Note that when you are using bufferevents, you do not need to call these IO functions; the bufferevents code does it for you.

The evbuffer\_write\_atmost() function was introduced in Libevent 2.0.1-alpha.

**Evbuffers and callbacks**

Users of evbuffers frequently want to know when data is added to or removed from an evbuffer. To support this, Libevent provides a generic evbuffer callback mechanism.

Interface

**struct** evbuffer\_cb\_info {

size\_t orig\_size;

size\_t n\_added;

size\_t n\_deleted;

};

**typedef void** (\*evbuffer\_cb\_func)(**struct** evbuffer\*buffer, **const struct** evbuffer\_cb\_info\*info, **void** \*arg);

An evbuffer callback is invoked whenever data is added to or removed from the evbuffer. It receives the buffer, a pointer to an evbuffer\_cb\_info structure, and a user-supplied argument. The evbuffer\_cb\_info structure’s orig\_size field records how many bytes there were on the buffer before its size changed; its n\_added field records how many bytes were added to the buffer, and its n\_deleted field records how many bytes were removed.

Interface

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 95 / 120 |

**struct** evbuffer\_cb\_entry;

**struct** evbuffer\_cb\_entry\*evbuffer\_add\_cb(**struct** evbuffer\*buffer,evbuffer\_cb\_func cb, **void** \*cbarg);

The evbuffer\_add\_cb() function adds a callback to an evbuffer, and returns an opaque pointer that can later be used to refer to this particular callback instance. The cb argument is the function that will be invoked, and the cbarg is the user-supplied pointer to pass to the function.

You can have multiple callbacks set on a single evbuffer. Adding a new callback does not remove old callbacks.

Example

#include <event2/buffer.h>

#include <stdio.h>

#include <stdlib.h>

/\* Here’s a callback that remembers how many bytes we have drained in total from the buffer, and prints a dot every time we hit a megabyte. \*/

**struct** total\_processed {

size\_t n;

};

**void** count\_megabytes\_cb(**struct** evbuffer\*buffer, **const struct** evbuffer\_cb\_info\*info, **void** \*arg)

{

**struct** total\_processed\*tp = arg;

size\_t old\_n = tp->n;

**int** megabytes, i;

tp->n += info->n\_deleted;

megabytes = ((tp->n) >> 20) - (old\_n >> 20);

**for** (i=0; i<megabytes; ++i)

putc(’.’, stdout);

}

**void** operation\_with\_counted\_bytes(**void**)

{

**struct** total\_processed\*tp = malloc(**sizeof**(\*tp)); **struct** evbuffer\*buf = evbuffer\_new();tp->n = 0;

evbuffer\_add\_cb(buf, count\_megabytes\_cb, tp);

/\* Use the evbuffer for a while. When we’re done: \*/ evbuffer\_free(buf);

free(tp);

}

Note in passing that freeing a nonempty evbuffer does not count as draining data from it, and that freeing an evbuffer does not free the user-supplied data pointer for its callbacks.

If you don’t want a callback to be permanently active on a buffer, you can remove it (to make it gone for good), or disable it (to turn it off for a while):

Interface

**int** evbuffer\_remove\_cb\_entry(**struct** evbuffer\*buffer,

**struct** evbuffer\_cb\_entry\*ent);

**int** evbuffer\_remove\_cb(**struct** evbuffer \*buffer, evbuffer\_cb\_func cb, **void** \*cbarg);

#define EVBUFFER\_CB\_ENABLED 1

**int** evbuffer\_cb\_set\_flags(**struct** evbuffer\*buffer, **struct** evbuffer\_cb\_entry\*cb,

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 96 / 120 |

ev\_uint32\_t flags);

**int** evbuffer\_cb\_clear\_flags(**struct** evbuffer\*buffer, **struct** evbuffer\_cb\_entry\*cb,ev\_uint32\_t flags);

You can remove a callback either by the evbuffer\_cb\_entry you got when you added it, or by the callback and pointer you used.

The evbuffer\_remove\_cb() functions return 0 on success and -1 on failure.

The evbuffer\_cb\_set\_flags() function and the evbuffer\_cb\_clear\_flags() function make a given flag be set or cleared on a given callback respectively. Right now, only one user-visible flag is supported: EVBUFFER\_CB\_ENABLED. The flag is set by default. When it is cleared, modifications to the evbuffer do not cause this callback to get invoked.

Interface

**int** evbuffer\_defer\_callbacks(**struct** evbuffer\*buffer, **struct** event\_base\*base);

As with bufferevent callbacks, you can cause evbuffer callbacks to not run immediately when the evbuffer is changed, but rather to be deferred and run as part of the event loop of a given event base. This can be helpful if you have multiple evbuffers whose callbacks potentially cause data to be added and removed from one another, and you want to avoid smashing the stack.

If an evbuffer’s callbacks are deferred, then when they are finally invoked, they may summarize the results for multiple operations.

Like bufferevents, evbuffers are internally reference-counted, so that it is safe to free an evbuffer even if it has deferred callbacks that have not yet executed.

This entire callback system was new in Libevent 2.0.1-alpha. The evbuffer\_cb\_(set|clear)\_flags() functions have existed with their present interfaces since 2.0.2-alpha.

**Avoiding data copies with evbuffer-based IO**

Really fast network programming often calls for doing as few data copies as possible. Libevent provides some mechanisms to help out with this.

Interface

**typedef void** (\*evbuffer\_ref\_cleanup\_cb)(**const void** \*data,size\_t datalen, **void** \*extra);

**int** evbuffer\_add\_reference(**struct** evbuffer\*outbuf,

**const void** \*data, size\_t datlen,

evbuffer\_ref\_cleanup\_cb cleanupfn, **void** \*extra);

This function adds a piece of data to the end of an evbuffer by reference. No copy is performed: instead, the evbuffer just stores a pointer to the datlen bytes stored at data. Therefore, the pointer must remain valid for as long as the evbuffer is using it. When the evbuffer no longer needs data, it will call the provided "cleanupfn" function with the provided "data" pointer, "datlen" value, and "extra" pointer as arguments. This function returns 0 on success, -1 on failure.

Example

#include <event2/buffer.h>

#include <stdlib.h>

#include <string.h>

/\* In this example, we have a bunch of evbuffers that we want to use to spool a one-megabyte resource out to the network. We do this without keeping any more copies of the resource in memory than necessary. \*/

#define HUGE\_RESOURCE\_SIZE (1024\*1024)

**struct** huge\_resource {

/\* We keep a count of the references that exist to this structure, so that we know when we can free it. \*/

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 97 / 120 |

**int** reference\_count;

**char** data[HUGE\_RESOURCE\_SIZE];

};

**struct** huge\_resource\*new\_resource(**void**) {

**struct** huge\_resource\*hr = malloc(**sizeof**(**struct** huge\_resource));hr->reference\_count = 1;

/\* Here we should fill hr->data with something. In real life, we’d probably load something or do a complex calculation. Here, we’ll just fill it with EEs. \*/

memset(hr->data, 0xEE, **sizeof**(hr->data));

**return** hr;

}

**void** free\_resource(**struct** huge\_resource\*hr) {

--hr->reference\_count;

**if** (hr->reference\_count == 0)

free(hr);

}

**static void** cleanup(**const void** \*data, size\_t len, **void** \*arg) {free\_resource(arg);

}

/\* This is the function that actually adds the resource to the buffer. \*/

**void** spool\_resource\_to\_evbuffer(**struct** evbuffer\*buf, **struct** huge\_resource\*hr)

{

++hr->reference\_count;

evbuffer\_add\_reference(buf, hr->data, HUGE\_RESOURCE\_SIZE, cleanup, hr);

}

The evbuffer\_add\_reference() function has had is present interface since 2.0.2-alpha.

**Adding a file to an evbuffer**

Some operating systems provide ways to write files to the network without ever copying the data to userspace. You can access these mechanisms, where available, with the simple interface:

Interface

**int** evbuffer\_add\_file(**struct** evbuffer\*output, **int** fd, ev\_off\_t offset,

size\_t length);

The evbuffer\_add\_file() function assumes that it has an open file descriptor (not a socket, for once!) fd that is available for reading. It adds length bytes from the file, starting at position offset, to the end of output. It returns 0 on success, or -1 on failure.

WARNING In Libevent 2.0.x, the only reliable thing to do with data added this way was to send it to the network with evbuffer\_write\*(), drain it with evbuffer\_drain(), or move it to another evbuffer with evbuffer\_\*\_buffer(). You couldn’t reli-ably extract it from the buffer with evbuffer\_remove(), linearize it with evbuffer\_pullup(), and so on. Libevent 2.1.x tries to fix this limitation.

If your operating system supports splice() or sendfile(), Libevent will use it to send data from fd to the network directly when call evbuffer\_write(), without copying the data to user RAM at all. If splice/sendfile don’t exist, but you have mmap(), Libevent will mmap the file, and your kernel can hopefully figure out that it never needs to copy the data to userspace. Otherwise, Libevent will just read the data from disk into RAM.

The file descriptor will be closed after the data is flushed from the evbuffer, or when the evbuffer is freed. If that’s not what you want, or if you want finer-grained control over the file, see the file\_segment functionality below.

This function was introduced in Libevent 2.0.1-alpha.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 98 / 120 |

**Fine-grained control with file segments**

The evbuffer\_add\_file() interface is inefficient for adding the same file more than once, since it takes ownership of the file.

Interface

**struct** evbuffer\_file\_segment;

**struct** evbuffer\_file\_segment\*evbuffer\_file\_segment\_new(

**int** fd, ev\_off\_t offset, ev\_off\_t length, **unsigned** flags); **void** evbuffer\_file\_segment\_free(**struct** evbuffer\_file\_segment\*seg); **int** evbuffer\_add\_file\_segment(**struct** evbuffer\*buf,

**struct** evbuffer\_file\_segment\*seg, ev\_off\_t offset, ev\_off\_t length);

The evbuffer\_file\_segment\_new() function creates and returns a new evbuffer\_file\_segment object to represent a piece of the underlying file stored in fd that begins at offset and contains length bytes. On error, it return NULL.

File segments are implemented with sendfile, splice, mmap, CreateFileMapping, or malloc()-and-read(), as appropriate. They’re created using the most lightweight supported mechanism, and transition to a heavier-weight mechanism as needed. (For example, if your OS supports sendfile and mmap, then a file segment can be implemented using only sendfile, until you try to actually inspect its contents. At that point, it needs to be mmap()ed.) You can control the fine-grained behavior of a file segment with these flags:

EVBUF\_FS\_CLOSE\_ON\_FREE

If this flag is set, freeing the file segment with evbuffer\_file\_segment\_free() will close the underlying file.

EVBUF\_FS\_DISABLE\_MMAP

If this flag is set, the file\_segment will never use a mapped-memory style backend (CreateFileMapping, mmap) for this file, even if that would be appropriate.

EVBUF\_FS\_DISABLE\_SENDFILE

If this flag is set, the file\_segment will never use a sendfile-style backend (sendfile, splice) for this file, even if that would be appropriate.

EVBUF\_FS\_DISABLE\_LOCKING

If this flag is set, no locks are allocated for the file segment: it won’t be safe to use it in any way where it can be seen by multiple threads.

Once you have an evbuffer\_file\_segment, you can add some or all of it to an evbuffer using evbuffer\_add\_file\_segment(). The offset argument here refers to an offset within the file segment, not to an offset within the file itself.

When you no longer want to use a file segment, you can free it with evbuffer\_file\_segment\_free(). The actual storage won’t be released until no evbuffer any longer holds a reference to a piece of the file segment.

Interface

**typedef void** (\*evbuffer\_file\_segment\_cleanup\_cb)(

**struct** evbuffer\_file\_segment **const** \*seg, **int** flags, **void** \*arg);

**void** evbuffer\_file\_segment\_add\_cleanup\_cb(**struct** evbuffer\_file\_segment\*seg,evbuffer\_file\_segment\_cleanup\_cb cb, **void** \*arg);

You can add a callback function to a file segment that will be invoked when the final reference to the file segment has been released and the file segment is about to get freed. This callback must not attempt to revivify the file segment, add it to any buffers, or so on.

These file-segment functions first appeared in Libevent 2.1.1-alpha; evbuffer\_file\_segment\_add\_cleanup\_cb() was added in 2.1.2-alpha.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 99 / 120 |

**Adding an evbuffer to another by reference**

You can also add one evbuffer’s to another by reference: rather than removing the contents of one buffer and adding them to another, you give one evbuffer a reference to another, and it behaves as though you had copied all the bytes in.

Interface

**int** evbuffer\_add\_buffer\_reference(**struct** evbuffer\*outbuf,

**struct** evbuffer\*inbuf);

The evbuffer\_add\_buffer\_reference() function behaves as though you had copied all the data from outbuf to inbuf, but does not perform any unnecessary copies. It returns 0 if successful and -1 on failure.

Note that subsequent changes to the contents of inbuf are not reflected in outbuf : this function adds the current contents of the evbuffer by reference, not the evbuffer itself.

Note also that you cannot nest buffer references: a buffer that has already been the outbuf of one evbuffer\_add\_buffer\_reference call cannot be the inbuf of another.

This function was introduced in Libevent 2.1.1-alpha.

**Making an evbuffer add- or remove-only**

Interface

**int** evbuffer\_freeze(**struct** evbuffer\*buf, **int** at\_front); **int** evbuffer\_unfreeze(**struct** evbuffer\*buf, **int** at\_front);

You can use these functions to temporarily disable changes to the front or end of an evbuffer. The bufferevent code uses them internally to prevent accidental modifications to the front of an output buffer, or the end of an input buffer.

The evbuffer\_freeze() functions were introduced in Libevent 2.0.1-alpha.

**Obsolete evbuffer functions**

The evbuffer interface changed a lot in Libevent 2.0. Before then, every evbuffers was implemented as a contiguous chunk of RAM, which made access very inefficient.

The event.h header used to expose the internals of struct evbuffer. These are no longer available; they changed too much between 1.4 and 2.0 for any code that relied on them to work.

To access the number of bytes in an evbuffer, there was an EVBUFFER\_LENGTH() macro. The actual data was available with EVBUFFER\_DATA(). These are both available in event2/buffer\_compat.h. Watch out, though: EVBUFFER\_DATA(b) is an alias for evbuffer\_pullup(b, -1), which can be very expensive.

Some other deprecated interfaces are:

Deprecated Interface

**char** \*evbuffer\_readline(**struct** evbuffer\*buffer); **unsigned char** \*evbuffer\_find(**struct** evbuffer\*buffer,

**const unsigned char** \*what, size\_t len);

The evbuffer\_readline() function worked like the current evbuffer\_readln(buffer, NULL, EVBUFFER\_EOL\_ANY).

The evbuffer\_find() function would search for the first occurrence of a string in a buffer, and return a pointer to it. Unlike evbuffer\_search(), it could only find the first string. To stay compatible with old code that uses this function, it now linearizes the entire buffer up to the end of the located string.

The callback interface was different too:

Deprecated Interface

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 100 / 120 |

**typedef void** (\*evbuffer\_cb)(**struct** evbuffer\*buffer,size\_t old\_len, size\_t new\_len, **void** \*arg);

**void** evbuffer\_setcb(**struct** evbuffer\*buffer, evbuffer\_cb cb, **void** \*cbarg);

An evbuffer could only have one callback set at a time, so setting a new callback would disable the previous callback, and setting a callback of NULL was the preferred way to disable a callbacks.

Instead of getting an evbuffer\_cb\_info\_structure, the function was called with the old and new lengths of the evbuffer. Thus, if old\_len was greater than new\_len, data was drained. If new\_len was greater than old\_len, data was added. It was not possible to defer callbacks, and so adds and deletes were never batched into a single callback invocation.

The obsolete functions here are still available in event2/buffer\_compat.h.

**Connection listeners: accepting TCP connections**

The evconnlistener mechanism gives you a way to listen for and accept incoming TCP connections.

All the functions and types in this section are declared in event2/listener.h. They first appeared in Libevent 2.0.2-alpha, unless otherwise noted.

**Creating or freeing an evconnlistener**

Interface

**struct** evconnlistener\*evconnlistener\_new(**struct** event\_base\*base,evconnlistener\_cb cb, **void** \*ptr, **unsigned** flags, **int** backlog,

evutil\_socket\_t fd);

**struct** evconnlistener\*evconnlistener\_new\_bind(**struct** event\_base\*base,evconnlistener\_cb cb, **void** \*ptr, **unsigned** flags, **int** backlog, **const struct** sockaddr\*sa, **int** socklen);

**void** evconnlistener\_free(**struct** evconnlistener\*lev);

The two evconnlistener\_new\*() functions both allocate and return a new connection listener object. A connection listener uses an event\_base to note when there is a new TCP connection on a given listener socket. When a new connection arrives, it invokes the callback function you give it.

In both functions, the base parameter is an event\_base that the listener should use to listen for connections. The cb function is a callback to invoke when a new connection is received; if cb is NULL, the listener is treated as disabled until a callback is set. The ptr pointer will be passed to the callback. The flags argument controls the behavior of the listener — more on this below. The backlog parameter controls the maximum number of pending connections that the network stack should allow to wait in a not-yet-accepted state at any time; see documentation for your system’s listen() function for more details. If backlog is negative, Libevent tries to pick a good value for the backlog; if it is zero, Libevent assumes that you have already called listen() on the socket you are providing it.

The functions differ in how they set up their listener socket. The evconnlistener\_new() function assumes that you have already bound a socket to the port you want to listen on, and that you’re passing the socket in as fd. If you want Libevent to allocate and bind to a socket on its own, call evconnlistener\_new\_bind(), and pass in the sockaddr you want to bind to, and its length.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 101 / 120 |

**Tip**

[When using evconnlistener\_new, make sure your listening socket is til\_make\_socket\_nonblocking or by manually setting the correct socket option. mode, undefined behavior might occur.]

in non-blocking mode by using evu-When the listening socket is left in blocking

To free a connection listener, pass it to evconnlistener\_free().

**Recognized flags**

These are the flags you can pass to the flags argument of the evconnlistener\_new() function. You can give any number of these, OR’d together.

LEV\_OPT\_LEAVE\_SOCKETS\_BLOCKING

By default, when the connection listener accepts a new incoming socket, it sets it up to be nonblocking so that you can use it with the rest of Libevent. Set this flag if you do not want this behavior.

LEV\_OPT\_CLOSE\_ON\_FREE

If this option is set, the connection listener closes its underlying socket when you free it.

LEV\_OPT\_CLOSE\_ON\_EXEC

If this option is set, the connection listener sets the close-on-exec flag on the underlying listener socket. See your platform documentation for fcntl and FD\_CLOEXEC for more information.

LEV\_OPT\_REUSEABLE

By default on some platforms, once a listener socket is closed, no other socket can bind to the same port until a while has passed. Setting this option makes Libevent mark the socket as reusable, so that once it is closed, another socket can be opened to listen on the same port.

LEV\_OPT\_THREADSAFE

Allocate locks for the listener, so that it’s safe to use it from multiple threads. New in Libevent 2.0.8-rc.

LEV\_OPT\_DISABLED

Initialize the listener to be disabled, not enabled. You can turn it on manually with evconnlistener\_enable(). New in Libevent 2.1.1-alpha.

LEV\_OPT\_DEFERRED\_ACCEPT

If possible, tell the kernel to not announce sockets as having been accepted until some data has been received on them, and they are ready for reading. Do not use this option if your protocol doesn’t start out with the client transmitting data, since in that case this option will sometimes cause the kernel to never tell you about the connection. Not all operating systems support this option: on ones that don’t, this option has no effect. New in Libevent 2.1.1-alpha.

**The connection listener callback**

Interface

**typedef void** (\*evconnlistener\_cb)(**struct** evconnlistener\*listener,evutil\_socket\_t sock, **struct** sockaddr \*addr, **int** len, **void** \*ptr);

When a new connection is received, the provided callback function is invoked. The listener argument is the connection listener that received the connection. The sock argument is the new socket itself. The addr and len arguments are the address from which the connection was received and the length of that address respectively. The ptr argument is the user-supplied pointer that was passed to evconnlistener\_new().

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 102 / 120 |

**Enabling and disabling an evconnlistener**

Interface

**int** evconnlistener\_disable(**struct** evconnlistener\*lev); **int** evconnlistener\_enable(**struct** evconnlistener\*lev);

These functions temporarily disable or reenable listening for new connections.

**Adjusting an evconnlistener’s callback**

Interface

**void** evconnlistener\_set\_cb(**struct** evconnlistener\*lev,

evconnlistener\_cb cb, **void** \*arg);

This function adjusts the callback and callback argument of an existing evconnlistener. It was introduced in 2.0.9-rc.

**Inspecting an evconnlistener**

Interface

evutil\_socket\_t evconnlistener\_get\_fd(**struct** evconnlistener \*lev); **struct** event\_base\*evconnlistener\_get\_base(**struct** evconnlistener\*lev);

These functions return a listener’s associated socket and event\_base respectively.

The evconnlistener\_get\_fd() function first appeared in Libevent 2.0.3-alpha.

**Detecting errors**

You can set an error callback that gets informed whenever an accept() call fails on the listener. This can be important to do if you’re facing an error condition that would lock the process unless you addressed it.

Interface

**typedef void** (\*evconnlistener\_errorcb)(**struct** evconnlistener\*lis, **void** \*ptr); **void** evconnlistener\_set\_error\_cb(**struct** evconnlistener\*lev,

evconnlistener\_errorcb errorcb);

If you use evconnlistener\_set\_error\_cb() to set an error callback on a listener, the callback will be invoked every time that an error occurs on the listener. It will receive the listener as its first argument, and the argument passed as ptr to evconnlistener\_new() as its second argument.

This function was introduced in Libevent 2.0.8-rc.

**Example code: an echo server.**

Example

#include <event2/listener.h>

#include <event2/bufferevent.h>

#include <event2/buffer.h>

#include <arpa/inet.h>

#include <string.h>

#include <stdlib.h>

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 103 / 120 |

#include <stdio.h>

#include <errno.h>

**static void**

echo\_read\_cb(**struct** bufferevent \*bev, **void** \*ctx)

{

/\* This callback is invoked when there is data to read on bev. \*/ **struct** evbuffer\*input = bufferevent\_get\_input(bev); **struct** evbuffer\*output = bufferevent\_get\_output(bev);

/\* Copy all the data from the input buffer to the output buffer. \*/ evbuffer\_add\_buffer(output, input);

}

**static void**

echo\_event\_cb(**struct** bufferevent \*bev, **short** events, **void** \*ctx)

{

**if** (events & BEV\_EVENT\_ERROR)

perror("Error from bufferevent");

**if** (events & (BEV\_EVENT\_EOF | BEV\_EVENT\_ERROR)) {bufferevent\_free(bev);

}

}

**static void**

accept\_conn\_cb(**struct** evconnlistener \*listener,

evutil\_socket\_t fd, **struct** sockaddr \*address, **int** socklen,

**void** \*ctx)

{

/\* We got a new connection! Set up a bufferevent for it. \*/ **struct** event\_base\*base = evconnlistener\_get\_base(listener); **struct** bufferevent\*bev = bufferevent\_socket\_new(

base, fd, BEV\_OPT\_CLOSE\_ON\_FREE);

bufferevent\_setcb(bev, echo\_read\_cb, NULL, echo\_event\_cb, NULL);

bufferevent\_enable(bev, EV\_READ|EV\_WRITE);

}

**static void**

accept\_error\_cb(**struct** evconnlistener \*listener, **void** \*ctx)

{

**struct** event\_base\*base = evconnlistener\_get\_base(listener); **int** err = EVUTIL\_SOCKET\_ERROR();

fprintf(stderr, "Got an error %d (%s) on the listener. "

"Shutting down.\n", err, evutil\_socket\_error\_to\_string(err));

event\_base\_loopexit(base, NULL);

}

**int**

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

{

**struct** event\_base\*base;

**struct** evconnlistener\*listener;

**struct** sockaddr\_in sin;

**int** port = 9876;

**if** (argc > 1) {

port = atoi(argv[1]);

}

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 104 / 120 |

**if** (port<=0 || port>65535) {

puts("Invalid port");

**return** 1;

}

base = event\_base\_new();

**if** (!base) {

puts("Couldn’t open event base");

**return** 1;

}

/\* Clear the sockaddr before using it, in case there are extra

* platform-specific fields that can mess us up. \*/ memset(&sin, 0, **sizeof**(sin));

/\* This is an INET address \*/ sin.sin\_family = AF\_INET;

/\* Listen on 0.0.0.0 \*/

sin.sin\_addr.s\_addr = htonl(0); /\* Listen on the given port. \*/ sin.sin\_port = htons(port);

listener = evconnlistener\_new\_bind(base, accept\_conn\_cb, NULL, LEV\_OPT\_CLOSE\_ON\_FREE|LEV\_OPT\_REUSEABLE, -1,

(**struct** sockaddr\*)&sin, **sizeof**(sin));

**if** (!listener) {

perror("Couldn’t create listener");

**return** 1;

}

evconnlistener\_set\_error\_cb(listener, accept\_error\_cb);

event\_base\_dispatch(base);

**return** 0;

}

**Using DNS with Libevent: high and low-level functionality**

Libevent provides a few APIs to use for resolving DNS names, and a facility for implementing simple DNS servers.

We’ll start by describing the higher-level facilities for name lookup, and then describe the low-level and server facilities.

Note There are known limitations in Libevent’s current DNS client implementation. It doesn’t support TCP lookups, DNSSec, or arbitrary record types. We’d like to fix all of these in some future version of Libevent, but for now, they’re not there.

**Preliminaries: Portable blocking name resolution**

To aid in porting programs that already use blocking name resolution, Libevent provides a portable implementation of the standard getaddrinfo() interface. This can be helpful when your program needs to run on platforms where either there is no getaddrinfo()

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 105 / 120 |

function, or where getaddrinfo() doesn’t conform to the standard as well as our replacement. (There are shockingly many of each.)

The getaddrinfo() interface is specified in RFC 3493, section 6.1. See the "Compatibility Notes" section below for a summary of how we fall short of a conformant implemenation.

Interface

**struct** evutil\_addrinfo {

**int** ai\_flags;

**int** ai\_family;

**int** ai\_socktype;

**int** ai\_protocol;

size\_t ai\_addrlen;

**char** \*ai\_canonname;

**struct** sockaddr\*ai\_addr;

**struct** evutil\_addrinfo\*ai\_next;

};

|  |  |  |
| --- | --- | --- |
| #define EVUTIL\_AI\_PASSIVE | /\* | ... \*/ |
| #define EVUTIL\_AI\_CANONNAME | /\* | ... \*/ |
| #define EVUTIL\_AI\_NUMERICHOST /\* | | ... \*/ |
| #define EVUTIL\_AI\_NUMERICSERV /\* | | ... \*/ |
| #define EVUTIL\_AI\_V4MAPPED | /\* | ... \*/ |
| #define EVUTIL\_AI\_ALL | /\* | ... \*/ |
| #define EVUTIL\_AI\_ADDRCONFIG | /\* | ... \*/ |
| **int** evutil\_getaddrinfo(**const char** \*nodename, **const char** \*servname, | | |
| **const struct** evutil\_addrinfo | | \*hints, **struct** evutil\_addrinfo \*\*res); |

**void** evutil\_freeaddrinfo(**struct** evutil\_addrinfo\*ai); **const char** \*evutil\_gai\_strerror(**int** err);

The evutil\_getaddrinfo() function tries to resolve the provided nodename and servname fields, according to the rules you give it in hints, and build you a linked list of evutil\_addrinfo structures and store them in \*res. It returns 0 on success, and a nonzero error code on failure.

You must provide at least one of nodename and servname. If nodename is provided, it is either a literal IPv4 address (like "127.0.0.1"), a literal IPv6 address (like "::1"), or a DNS name (like "www.example.com"). If servname is provided, it is either the symbolic name of a network service (like "https") or a string containing a port number given in decimal (like "443").

If you do not specify servname, then the port values in \*res will be set to zero. If you do not specify nodename, then the addresses in \*res will either be for localhost (by default), or for "any" (if EVUTIL\_AI\_PASSIVE is set.)

The ai\_flags field of hints tells evutil\_getaddrinfo how to perform the lookup. It can contain zero or more of the flags below, ORed together.

EVUTIL\_AI\_PASSIVE

This flag indicates that we’re going to be using the address for listening, not for connection. Ordinarily this makes no difference, except when nodename is NULL: for connecting, a NULL nodename is localhost (127.0.0.1 or ::1), whereas when listening, a NULL node name is ANY (0.0.0.0 or ::0).

EVUTIL\_AI\_CANONNAME

If this flag is set, we try to report the canonical name for the host in the ai\_canonname field.

EVUTIL\_AI\_NUMERICHOST

When this flag is set, we only resolve numeric IPv4 and IPv6 addresses; if the nodename would require a name lookup, we instead give an EVUTIL\_EAI\_NONAME error.

EVUTIL\_AI\_NUMERICSERV

When this flag is set, we only resolve numeric service names. If the servname is neither NULL nor a decimal integer, give an EVUTIL\_EAI\_NONAME error.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 106 / 120 |

EVUTIL\_AI\_V4MAPPED

This flag indicates that if ai\_family is AF\_INET6, and no IPv6 addresses are found, any IPv4 addresses in the result should be returned as v4-mapped IPv6 addresses. It is not currently supported by evutil\_getaddrinfo() unless the OS supports it.

EVUTIL\_AI\_ALL

If this flag and EVUTIL\_AI\_V4MAPPED are both set, then IPv4 addresses in the result included in the result as 4-mapped IPv6 addresses, whether there are any IPv6 addresses or not. It is not currently supported by evutil\_getaddrinfo() unless the OS supports it.

EVUTIL\_AI\_ADDRCONFIG

If this flag is set, then IPv4 addresses are only included in the result if the system has a nonlocal IPv4 address, and IPv6 addresses are only included in the result if the system has a nonlocal IPv6 address.

The ai\_family field of hints is used to tell evutil\_getaddrinfo() which addresses it should return. It can be AF\_INET to request IPv4 addresses only, AF\_INET6 to request IPv6 addresses only, or AF\_UNSPEC to request all available addresses.

The ai\_socktype and ai\_protocol fields of hints are used to tell evutil\_getaddrinfo() how you’re going to use the address. They’re the same as the socktype and protocol fields you would pass to socket().

If evutil\_getaddrinfo() is successful, it allocates a new linked list of evutil\_addrinfo structures, where each points to the next with its "ai\_next" pointer, and stores them in \*res. Because this value is heap-allocated, you will need to use evutil\_freeaddrinfo to free it.

If it fails, it returns one of these numeric error codes:

EVUTIL\_EAI\_ADDRFAMILY

You requested an address family that made no sense for the nodename.

EVUTIL\_EAI\_AGAIN

There was a recoverable error in name resolution; try again later.

EVUTIL\_EAI\_FAIL

There was a non-recoverable error in name resolution; your resolver or your DNS server may be busted.

EVUTIL\_EAI\_BADFLAGS

The ai\_flags field in hints was somehow invalid.

EVUTIL\_EAI\_FAMILY

The ai\_family field in hints was not one we support.

EVUTIL\_EAI\_MEMORY

We ran out of memory while trying to answer your request.

EVUTIL\_EAI\_NODATA

The host you asked for exists, but has no address information associated with it. (Or, it has no address information of the type you requested.)

EVUTIL\_EAI\_NONAME

The host you asked for doesn’t seem to exist.

EVUTIL\_EAI\_SERVICE

The service you asked for doesn’t seem to exist.

EVUTIL\_EAI\_SOCKTYPE

We don’t support the socket type you asked for, or it isn’t compatible with ai\_protocol.

EVUTIL\_EAI\_SYSTEM

There was some other system error during name resolution. Check errno for more information.

EVUTIL\_EAI\_CANCEL

The application requested that this DNS lookup should be canceled before it was finished. The evutil\_getaddrinfo() func-tion never produces this error, but it can come from evdns\_getaddrinfo() as described in the section below.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 107 / 120 |

You can use evutil\_gai\_strerror() to convert one of these results into a human-readable string.

Note: If your OS defines struct addrinfo, then evutil\_addrinfo is just an alias for your OS’s built-in structure. Similarly, if your operating system defines any of the AI\_\* flags, then the corresponding EVUTIL\_AI\_\* flag is just an alias for the native flag; and if your operating system defines any of the EAI\_\* errors, then the corresponding EVUTIL\_EAI\_\* code is the same as your platform’s native error code.

Example: Resolving a hostname and making a blocking connection

#include <event2/util.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <stdio.h>

#include <string.h>

#include <assert.h>

#include <unistd.h>

evutil\_socket\_t

get\_tcp\_socket\_for\_host(**const char** \*hostname, ev\_uint16\_t port)

{

**char** port\_buf[6];

**struct** evutil\_addrinfo hints;

**struct** evutil\_addrinfo\*answer = NULL;

**int** err;

evutil\_socket\_t sock;

/\* Convert the port to decimal. \*/

evutil\_snprintf(port\_buf, **sizeof**(port\_buf), "%d", (**int**)port);

/\* Build the hints to tell getaddrinfo how to act. \*/ memset(&hints, 0, **sizeof**(hints));

hints.ai\_family = AF\_UNSPEC; /\* v4 or v6 is fine. \*/ hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP; /\* We want a TCP socket \*/ /\* Only return addresses we can use. \*/

hints.ai\_flags = EVUTIL\_AI\_ADDRCONFIG;

/\* Look up the hostname. \*/

err = evutil\_getaddrinfo(hostname, port\_buf, &hints, &answer); **if** (err != 0) {

fprintf(stderr, "Error **while** resolving ’%s’: %s", hostname, evutil\_gai\_strerror(err));

**return** -1;

}

/\* If there was no error, we should have at least one answer. \*/ assert(answer);

/\* Just use the first answer. \*/

sock = socket(answer->ai\_family,

answer->ai\_socktype,

answer->ai\_protocol);

**if** (sock < 0)

**return** -1;

**if** (connect(sock, answer->ai\_addr, answer->ai\_addrlen)) {

/\* Note that we’re doing a blocking connect in this function.

* If this were nonblocking, we’d need to treat some errors
* (like EINTR and EAGAIN) specially. \*/

EVUTIL\_CLOSESOCKET(sock);

**return** -1;

}

**return** sock;

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 108 / 120 |

}

These functions and constants were new in Libevent 2.0.3-alpha. They are declared in event2/util.h.

**Non-blocking hostname resolution with evdns\_getaddrinfo()**

The main problem with the regular getaddrinfo() interface, and with evutil\_getaddrinfo() above, is that they’re blocking: when you call them, the thread you’re in has to wait while they query your DNS server(s) and wait for a response. Since you’re using Libevent, that probably isn’t the behavior you want.

So for nonblocking use, Libevent provides a set of functions to launch DNS requests, and use Libevent to wait for the server to answer.

Interface

**typedef void** (\*evdns\_getaddrinfo\_cb)(

**int** result, **struct** evutil\_addrinfo\*res, **void** \*arg); **struct** evdns\_getaddrinfo\_request;

**struct** evdns\_getaddrinfo\_request\*evdns\_getaddrinfo( **struct** evdns\_base\*dns\_base,

**const char** \*nodename, **const char** \*servname,

**const struct** evutil\_addrinfo\*hints\_in,

evdns\_getaddrinfo\_cb cb, **void** \*arg);

**void** evdns\_getaddrinfo\_cancel(**struct** evdns\_getaddrinfo\_request\*req);

The evdns\_getaddrinfo() function behaves just like evutil\_getaddrinfo(), except that instead of blocking on DNS servers, it uses Libevent’s low-level DNS facilities to look hostnames up for you. Because it can’t always return you the result immediately, you need to provide it a callback function of type evdns\_getaddrinfo\_cb, and an optional user-supplied argument for that callback function.

Additionally, you need to provide evdns\_getaddrinfo() with a pointer to an evdns\_base. This structure holds the state and configuration for Libevent’s DNS resolver. See the next section for more information on how to get one.

The evdns\_getaddrinfo() function returns NULL if it fails or succeeds immediately. Otherwise, it returns a pointer to an evdns\_getaddrinfo\_request. You can use this to cancel the request with evdns\_getaddrinfo\_cancel() at any time before the request is finished.

Note that the callback function will eventually be invoked whether evdns\_getaddrinfo() returns NULL or not, and whether evdns\_getaddrinfo\_cancel() is called or not.

When you call evdns\_getaddrinfo(), it makes its own internal copies of its nodename, servname, and hints arguments: you do not need to ensure that they continue to exist while the name lookup is in progress.

Example: Nonblocking lookups with evdns\_getaddrinfo()

#include <event2/dns.h>

#include <event2/util.h>

#include <event2/event.h>

#include <sys/socket.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <assert.h>

**int** n\_pending\_requests = 0;

**struct** event\_base\*base = NULL;

**struct** user\_data {

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Fast portable non-blocking network programming with Libevent | | | | 109 / 120 | |
|  |  |  |  |  |  |
|  | **char** \*name; | /\* the name we’re resolving | \*/ |  |  |
|  | **int** idx; /\* | its position on the command | line \*/ |  |  |
| }; | |  |  |  |  |

**void** callback(**int** errcode, **struct** evutil\_addrinfo\*addr, **void** \*ptr)

{

**struct** user\_data\*data = ptr;

**const char** \*name = data->name;

**if** (errcode) {

printf("%d. %s -> %s\n", data->idx, name, evutil\_gai\_strerror(errcode));

} **else** {

**struct** evutil\_addrinfo\*ai;

printf("%d. %s", data->idx, name);

**if** (addr->ai\_canonname)

printf(" [%s]", addr->ai\_canonname);

puts("");

**for** (ai = addr; ai; ai = ai->ai\_next) {

**char** buf[128];

**const char** \*s = NULL;

**if** (ai->ai\_family == AF\_INET) {

**struct** sockaddr\_in\*sin = (**struct** sockaddr\_in\*)ai->ai\_addr;s = evutil\_inet\_ntop(AF\_INET, &sin->sin\_addr, buf, 128);

} **else if** (ai->ai\_family == AF\_INET6) {

**struct** sockaddr\_in6\*sin6 = (**struct** sockaddr\_in6\*)ai->ai\_addr;s = evutil\_inet\_ntop(AF\_INET6, &sin6->sin6\_addr, buf, 128);

}

**if** (s)

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

}

evutil\_freeaddrinfo(addr);

}

free(data->name);

free(data);

**if** (--n\_pending\_requests == 0)

event\_base\_loopexit(base, NULL);

}

/\* Take a list of domain names from the command line and resolve them in \* parallel. \*/

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

{

**int** i;

**struct** evdns\_base\*dnsbase;

**if** (argc == 1) {

puts("No addresses given.");

**return** 0;

}

base = event\_base\_new();

**if** (!base)

**return** 1;

dnsbase = evdns\_base\_new(base, 1);

**if** (!dnsbase)

**return** 2;

**for** (i = 1; i < argc; ++i) {

**struct** evutil\_addrinfo hints;

**struct** evdns\_getaddrinfo\_request\*req;

**struct** user\_data\*user\_data;

memset(&hints, 0, **sizeof**(hints));

hints.ai\_family = AF\_UNSPEC;

hints.ai\_flags = EVUTIL\_AI\_CANONNAME;

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 110 / 120 |

/\* Unless we specify a socktype, we’ll get at least two entries for

* each address: one for TCP and one for UDP. That’s not what we
* want. \*/

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP;

**if** (!(user\_data = malloc(**sizeof**(**struct** user\_data)))) {perror("malloc");

exit(1);

}

**if** (!(user\_data->name = strdup(argv[i]))) {

perror("strdup");

exit(1);

}

user\_data->idx = i;

++n\_pending\_requests;

req = evdns\_getaddrinfo(

dnsbase, argv[i], NULL /\* no service name given \*/, &hints, callback, user\_data);

**if** (req == NULL) {

printf(" [request **for** %s returned immediately]\n", argv[i]);

/\* No need to free user\_data or decrement n\_pending\_requests; that \* happened in the callback. \*/

}

}

**if** (n\_pending\_requests)

event\_base\_dispatch(base);

evdns\_base\_free(dnsbase, 0);

event\_base\_free(base);

**return** 0;

}

These functions were new in Libevent 2.0.3-alpha. They are declared in event2/dns.h.

**Creating and configuring an evdns\_base**

Before you can do nonblocking DNS lookups with evdns, you’ll need to configure an evdns\_base. Each evdns\_base stores a list of nameservers, and DNS configuration options, and tracks active and in-flight DNS requests.

Interface

**struct** evdns\_base \*evdns\_base\_new(**struct** event\_base \*event\_base, **int** initialize);

**void** evdns\_base\_free(**struct** evdns\_base\*base, **int** fail\_requests);

The evdns\_base\_new() function returns a new evdns\_base on success, and NULL on failure. If the initialize argument is 1, it tries to configure the DNS base sensibly given your operating system’s default. If it is 0, it leaves the evdns\_base empty, with no nameservers or options configured.

When you no longer need an evdns\_base, you can free it with evdns\_base\_free. If its fail\_requests argument is true, it will make all in-flight requests get their callbacks invoked with a canceled error code before it frees the base.

**Initializing evdns from the system configuration**

If you want a little more control over how the evdns\_base is initialized, you can pass 0 as the initialize argument to evdns\_base\_new, and invoke one of these functions.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 111 / 120 |

Interface

#define DNS\_OPTION\_SEARCH 1

#define DNS\_OPTION\_NAMESERVERS 2

#define DNS\_OPTION\_MISC 4

#define DNS\_OPTION\_HOSTSFILE 8

#define DNS\_OPTIONS\_ALL 15

**int** evdns\_base\_resolv\_conf\_parse(**struct** evdns\_base\*base, **int** flags, **const char** \*filename);

#ifdef WIN32

**int** evdns\_base\_config\_windows\_nameservers(**struct** evdns\_base\*);#define EVDNS\_BASE\_CONFIG\_WINDOWS\_NAMESERVERS\_IMPLEMENTED #endif

The evdns\_base\_resolv\_conf\_parse() function will scan the resolv.conf formatted file stored in filename, and read in all the options from it that are listed in flags. (For more information on the resolv.conf file, see your local Unix manual pages.)

DNS\_OPTION\_SEARCH

Tells evdns to read the domain and search fields from the resolv.conf file and the ndots option, and use them to decide which domains (if any) to search for hostnames that aren’t fully-qualified.

DNS\_OPTION\_NAMESERVERS

This flag tells evdns to learn the nameservers from the resolv.conf file.

DNS\_OPTION\_MISC

Tells evdns to set other configuration options from the resolv.conf file.

DNS\_OPTION\_HOSTSFILE

Tells evdns to read a list of hosts from /etc/hosts as part of loading the resolv.conf file.

DNS\_OPTIONS\_ALL

Tells evdns to learn as much as it can from the resolv.conf file.

On Windows, you don’t have a resolv.conf file to tell you where your nameservers are, so you can use the evdns\_base\_config\_windows\_n function to read all your nameservers from your registry (or your NetworkParams, or wherever they’re hidden).

**The resolv.conf file format**

The resolv.conf format we recognize is a text file, each line of which should either be empty, contain a comment starting with the # character, or consist of a token followed zero or more arguments. The tokens we recognize are:

nameserver

Must be followed by the IP address of exactly one nameserver. As an extension, Libevent allows you to specify a nonstan-dard port for the nameserver, using the IP:Port or the [IPv6]:port syntax.

domain

The local domain name.

search

A list of names to search when resolving local hostnames. Any name that has fewer than "ndots" dots in it is considered local, and if we can’t resolve it as-is, we look in these domain names. For example, if "search" is example.com and "ndots" is 1, then when the user asks us to resolve "www", we will consider "www.example.com".

options

A space-separated list of options. Each option is given either as a bare string, or (if it takes an argument) in the option:value format. Recognized options are:

ndots:INTEGER

Used to configure searching. See "search" above. Defaults to 1.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 112 / 120 |

timeout:FLOAT

How long, in seconds, do we wait for a response from a DNS server before we assume we aren’t getting one? Defaults to 5 seconds.

max-timeouts:INT

How many times do we allow a nameserver to time-out in a row before we assume that it’s down? Defaults to 3.

max-inflight:INT

How many DNS requests do we allow to be pending at once? (If we try to do more requests than this, the extras will stall until the earlier ones are answered or time out.) Defaults to 64.

attempts:INT

How many times to we re-transmit a DNS request before giving up on it? Defaults to 3.

randomize-case:INT

If nonzero, we randomize the case on outgoing DNS requests and make sure that replies have the same case as our requests. This so-called "0x20 hack" can help prevent some otherwise simple active events against DNS. Defaults to 1.

bind-to:ADDRESS

If provided, we bind to the given address whenever we send packets to a nameserver. As of Libevent 2.0.4-alpha, it only applied to subsequent nameserver entries.

initial-probe-timeout:FLOAT

When we decide that a nameserver is down, we probe it with exponentially decreasing frequency to see if it has come back up. This option configures the first timeout in the series, in seconds. Defaults to 10.

getaddrinfo-allow-skew:FLOAT

When evdns\_getaddrinfo() requests both an IPv4 address and an IPv6 address, it does so in separate DNS request packets, since some servers can’t handle both requests in one packet. Once it has an answer for one address type, it waits a little while to see if an answer for the other one comes in. This option configures how long to wait, in seconds. Defaults to 3 seconds.

Unrecognized tokens and options are ignored.

**Configuring evdns manually**

If you want even more fine-grained control over evdns’s behavior, you can use these functions:

Interface

**int** evdns\_base\_nameserver\_sockaddr\_add(**struct** evdns\_base\*base,

**const struct** sockaddr\*sa, ev\_socklen\_t len,

**unsigned** flags);

**int** evdns\_base\_nameserver\_ip\_add(**struct** evdns\_base\*base, **const char** \*ip\_as\_string);

**int** evdns\_base\_load\_hosts(**struct** evdns\_base\*base, **const char** \*hosts\_fname);

**void** evdns\_base\_search\_clear(**struct** evdns\_base\*base);

**void** evdns\_base\_search\_add(**struct** evdns\_base\*base, **const char** \*domain); **void** evdns\_base\_search\_ndots\_set(**struct** evdns\_base\*base, **int** ndots);

**int** evdns\_base\_set\_option(**struct** evdns\_base \*base, **const char** \*option, **const char** \*val);

**int** evdns\_base\_count\_nameservers(**struct** evdns\_base\*base);

The evdns\_base\_nameserver\_sockaddr\_add() function adds a nameserver to an existing evdns\_base by its address. The flags argument is currently ignored, and should be 0 for forward-compatibility. The function returns 0 on success and negative on failure. (It was added in Libevent 2.0.7-rc.)

The evdns\_base\_nameserver\_ip\_add function adds a nameserver to an existing evdns\_base. It takes the nameserver in a text string, either as an IPv4 address, an IPv6 address, an IPv4 address with a port (IPv4:Port), or an IPv6 address with a port ([IPv6]:Port). It returns 0 on success and negative on failure.

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 113 / 120 |

The evdns\_base\_load\_hosts() function loads a hosts file (in the same format as /etc/hosts) from hosts\_fname. It also returns 0 on success and negative on failure.

The evdns\_base\_search\_clear() function removes all current search suffixes (as configured by the search option) from the evdns\_base; the evdns\_base\_search\_add() function adds a suffix.

The evdns\_base\_set\_option() function sets a given option to a given value in the evdns\_base. Each one is given as a string. (Before Libevent 2.0.3, the option name needed to have a colon after it.)

If you’ve just parsed a set of configuration files and want to see if any nameservers were added, you can use evdns\_base\_count\_nameserve to see how many there are.

**Library-side configuration**

There are a couple of functions you can use to specify library-wide settings for the evdns module:

Interface

**typedef void** (\*evdns\_debug\_log\_fn\_type)(**int** is\_warning, **const char** \*msg); **void** evdns\_set\_log\_fn(evdns\_debug\_log\_fn\_type fn);

**void** evdns\_set\_transaction\_id\_fn(ev\_uint16\_t (\*fn)(**void**));

For historical reasons, the evdns subsystem does its own logging; you can use evdns\_set\_log\_fn() to give it a callback that does something with its messages besides discard them.

For security, evdns needs a good source of random numbers: it uses this to pick hard-to-guess transaction IDs and to randomize queries when using the 0x20 hack. (See the "randomize-case" option for more info here.) Older versions of Libevent, did not pro-vide a secure RNG of its own, however. You can give evdns a better random number generator by calling evdns\_set\_transaction\_id\_fn and giving it a function that returns a hard-to-predict two-byte unsigned integer.

In Libevent 2.0.4-alpha and later, Libevent uses its own built-in secure RNG; evdns\_set\_transaction\_id\_fn() has no effect.

**Low-level DNS interfaces**

Occasionally, you’ll want the ability to launch specific DNS requests with more fine-grained control than you get from evdns\_getaddrinfo Libevent gives you some interfaces to do that.

Missing features Right now, Libevent’s DNS support lacks a few features that you’d expect from a low-level DNS system, like support for arbitrary request types and TCP requests. If you need features that evdns doesn’t have, please consider contributing a patch. You might also look into a more full-featured DNS library like c-ares.

Interface

#define DNS\_QUERY\_NO\_SEARCH /\* ... \*/

#define DNS\_IPv4\_A #define DNS\_PTR #define DNS\_IPv6\_AAAA

/\* ... \*/

/\* ... \*/

/\* ... \*/

**typedef void** (\*evdns\_callback\_type)(**int** result, **char** type, **int** count, **int** ttl, **void** \*addresses, **void** \*arg);

**struct** evdns\_request\*evdns\_base\_resolve\_ipv4(**struct** evdns\_base\*base, **const char** \*name, **int** flags, evdns\_callback\_type callback, **void** \*ptr);

**struct** evdns\_request\*evdns\_base\_resolve\_ipv6(**struct** evdns\_base\*base, **const char** \*name, **int** flags, evdns\_callback\_type callback, **void** \*ptr);

**struct** evdns\_request\*evdns\_base\_resolve\_reverse(**struct** evdns\_base\*base, **const struct** in\_addr\*in, **int** flags, evdns\_callback\_type callback, **void** \*ptr);

**struct** evdns\_request\*evdns\_base\_resolve\_reverse\_ipv6(

**struct** evdns\_base\*base, **const struct** in6\_addr\*in, **int** flags,evdns\_callback\_type callback, **void** \*ptr);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 114 / 120 |

These resolve functions initiate a DNS request for a particular record. Each takes an evdns\_base to use for the request, a resource to look up (either a hostname for forward lookups, or an address for reverse lookups), a set of flags to determine how to do the lookup, a callback to invoke when the lookup is done, and a pointer to pass to the user-supplied callback.

The flags argument is either 0 or DNS\_QUERY\_NO\_SEARCH to explicitly suppress searching in the list of search if the original search fails. DNS\_QUERY\_NO\_SEARCH has no effect for reverse lookups, since those never do searching.

When the request is done---either successfully or not---the callback function will be invoked. The callback takes a result that indicates success or an error code (see DNS Errors table below), a record type (one of DNS\_IPv4\_A, DNS\_IPv6\_AAAA, or DNS\_PTR), the number of records in addresses, a time-to-live in seconds, the addresses themselves, and the user-supplied argument pointer.

The addresses argument to the callback is NULL in the event of an error. For a PTR record, it’s a NUL-terminated string. For IPv4 records, it is an array of four-byte values in network order. For IPv6 records, it is an array of 16-byte records in network order. (Note that the number of addresses can be 0 even if there was no error. This can happen when the name exists, but it has no records of the requested type.)

The errors codes that can be passed to the callback are as follows:

DNS Errors [options="header",width="70%"

|  |  |
| --- | --- |
| Code | Meaning |
| DNS\_ERR\_NONE | No error occurred |
| DNS\_ERR\_FORMAT | The server didn’t understand the query |
| DNS\_ERR\_SERVERFAILED | The server reported an internal error |
| DNS\_ERR\_NOTEXIST | There was no record with the given name |
| DNS\_ERR\_NOTIMPL | The server doesn’t understand this kind of query |
| DNS\_ERR\_REFUSED | The server rejected the query for policy reasons |
| DNS\_ERR\_TRUNCATED | The DNS record wouldn’t fit in a UDP packet |
| DNS\_ERR\_UNKNOWN | Unknown internal error |
| DNS\_ERR\_TIMEOUT | We waited too long for an answer |
| DNS\_ERR\_SHUTDOWN | The user asked us to shut down the evdns system |
| DNS\_ERR\_CANCEL | The user asked us to cancel this request |
| DNS\_ERR\_NODATA | The response arrived, but contained no answers |

(DNS\_ERR\_NODATA was new in 2.0.15-stable.)

You can decode these error codes to a human-readable string with:

Interface

**const char** \*evdns\_err\_to\_string(**int** err);

Each resolve function returns a pointer to an opaque evdns\_request structure. You can use this to cancel the request at any point before the callback is invoked:

Interface

**void** evdns\_cancel\_request(**struct** evdns\_base\*base,

**struct** evdns\_request\*req);

Canceling a request with this function makes its callback get invoked with the DNS\_ERR\_CANCEL result code.

**Suspending DNS client operations and changing nameservers**

Sometimes you want to reconfigure or shut down the DNS subsystem without affecting in-flight DNS request too much. Interface

**int** evdns\_base\_clear\_nameservers\_and\_suspend(**struct** evdns\_base\*base); **int** evdns\_base\_resume(**struct** evdns\_base\*base);

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 115 / 120 |

If you call evdns\_base\_clear\_nameservers\_and\_suspend() on an evdns\_base, all nameservers are removed, and pending requests are left in limbo until later you re-add nameservers and call evdns\_base\_resume().

These functions return 0 on success and -1 on failure. They were introduced in Libevent 2.0.1-alpha.

**DNS server interfaces**

Libevent provides simple functionality for acting as a trivial DNS server and responding to UDP DNS requests.

This section assumes some familiarity with the DNS protocol.

**Creating and closing a DNS server**

Interface

**struct** evdns\_server\_port\*evdns\_add\_server\_port\_with\_base( **struct** event\_base\*base,

evutil\_socket\_t socket,

**int** flags,

evdns\_request\_callback\_fn\_type callback,

**void** \*user\_data);

**typedef void** (\*evdns\_request\_callback\_fn\_type)(

**struct** evdns\_server\_request\*request,

**void** \*user\_data);

**void** evdns\_close\_server\_port(**struct** evdns\_server\_port\*port);

To begin listening for DNS requests, call evdns\_add\_server\_port\_with\_base(). It takes an event\_base to use for event handling; a UDP socket to listen on; a flags variable (always 0 for now); a callback function to call when a new DNS query is received; and a pointer to user data that will be passed to the callback. It returns a new evdns\_server\_port object.

When you are done with the DNS server, you can pass it to evdns\_close\_server\_port().

The evdns\_add\_server\_port\_with\_base() function was new in 2.0.1-alpha; evdns\_close\_server\_port() was introduced in 1.3.

**Examining a DNS request**

Unfortunately, Libevent doesn’t currently provide a great way to look at DNS requests via a programmatic interface. Instead, you’re stuck including event2/dns\_struct.h and looking at the evdns\_server\_request structure manually.

It would be great if a future version of Libevent provided a better way to do this.

Interface

**struct** evdns\_server\_request {

**int** flags;

**int** nquestions;

**struct** evdns\_server\_question\*\*questions;

};

#define EVDNS\_QTYPE\_AXFR 252

#define EVDNS\_QTYPE\_ALL 255

**struct** evdns\_server\_question {

**int** type;

**int** dns\_question\_class;

**char** name[1];

};

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 116 / 120 |

The flags field of the request contains the DNS flags set in the request; the nquestions field is the number of questions in the request; and questions is an array of pointers to struct evdns\_server\_question. Each evdns\_server\_question includes the resource type of the request (see below for a list of EVDNS\_\*\_TYPE macros), the class of the request (typically EVDNS\_CLASS\_INET), and the name of the requested hostname.

These structures were introduced in Libevent 1.3. Before Libevent 1.4, dns\_question\_class was called "class", which made trouble for the C++ people. C programs that still use the old "class" name will stop working in a future release.

Interface

**int** evdns\_server\_request\_get\_requesting\_addr(**struct** evdns\_server\_request\*req,

**struct** sockaddr\*sa, **int** addr\_len);

Sometimes you’ll want to know which address made a particular DNS request. You can check this by calling evdns\_server\_request\_get\_r on it. You should pass in a sockaddr with enough storage to hold the address: struct sockaddr\_storage is recommended.

This function was introduced in Libevent 1.3c.

**Responding to DNS requests**

Every time your DNS server receives a request, the request is passed to the callback function you provided, along with your user\_data pointer. The callback function must either respond to the request, ignore the request, or make sure that the request is eventually answered or ignored.

Before you respond to a request, you can add one or more answers to your response:

Interface

**int** evdns\_server\_request\_add\_a\_reply(**struct** evdns\_server\_request \*req, **const char** \*name, **int** n, **const void** \*addrs, **int** ttl);

**int** evdns\_server\_request\_add\_aaaa\_reply(**struct** evdns\_server\_request \*req, **const char** \*name, **int** n, **const void** \*addrs, **int** ttl);

**int** evdns\_server\_request\_add\_cname\_reply(**struct** evdns\_server\_request \*req, **const char** \*name, **const char** \*cname, **int** ttl);

The functions above all add a single RR (of type A, AAAA, or CNAME respectively) to the answers section of a DNS reply for the request req. In each case the argument name is the hostname to add an answer for, and ttl is the time-to-live value of the answer in seconds. For A and AAAA records, n is the number of addresses to add, and addrs is a pointer to the raw addresses, either given as a sequence of n\*4 bytes for IPv4 addresses in an A record, or as a sequence of n\*16 bytes for IPv6 addresses in an AAAA record.

These functions return 0 on success and -1 on failure.

Interface

**int** evdns\_server\_request\_add\_ptr\_reply(**struct** evdns\_server\_request \*req, **struct** in\_addr\*in, **const char** \*inaddr\_name, **const char** \*hostname,

**int** ttl);

This function adds a PTR record to the answer section of a request. The arguments req and ttl are as above. You must provide exactly one of in (an IPv4 address) or inaddr\_name (an address in the .arpa domain) to indicate which address you’re providing a response for. The hostname argument is the answer for the PTR lookup.

Interface

#define EVDNS\_ANSWER\_SECTION 0

#define EVDNS\_AUTHORITY\_SECTION 1

#define EVDNS\_ADDITIONAL\_SECTION 2

|  |  |  |
| --- | --- | --- |
| #define EVDNS\_TYPE\_A | | 1 |
| #define EVDNS\_TYPE\_NS | | 2 |
| #define | EVDNS\_TYPE\_CNAME | 5 |
| #define | EVDNS\_TYPE\_SOA | 6 |
|  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Fast portable non-blocking network programming with Libevent | | | 117 / 120 | |
|  |  |  |  |  |  |
|  | #define | EVDNS\_TYPE\_PTR | 12 |  |  |
|  | #define | EVDNS\_TYPE\_MX | 15 |  |  |
|  | #define | EVDNS\_TYPE\_TXT | 16 |  |  |
|  | #define | EVDNS\_TYPE\_AAAA | 28 |  |  |
|  | #define | EVDNS\_CLASS\_INET 1 | |  |  |
|  | **int** evdns\_server\_request\_add\_reply(**struct** evdns\_server\_request\*req, | | |  |  |
|  | **int** | section, **const char** \*name, **int** type, **int** dns\_class, **int** ttl, | |  |  |
|  | **int** | datalen, **int** is\_name, **const char** \*data); | |  |  |
|  |  |  |  |  |  |

This function adds an arbitrary RR to the DNS reply of a request req. The section argument describes which section to add it to, and should be one of the EVDNS\_\*\_SECTION values. The name argument is the name field of the RR. The type argument is the type field of the RR, and should be one of the EVDNS\_TYPE\_\* values if possible. The dns\_class argument is the class field of the RR, and should generally be EVDNS\_CLASS\_INET. The ttl argument is the time-to-live in seconds of the RR. The rdata and rdlength fields of the RR will be generated from the datalen bytes provided in data. If is\_name is true, the data will be encoded as a DNS name (i.e., with DNS name compression). Otherwise, it’s included verbatim.

Interface

**int** evdns\_server\_request\_respond(**struct** evdns\_server\_request\*req, **int** err); **int** evdns\_server\_request\_drop(**struct** evdns\_server\_request\*req);

The evdns\_server\_request\_respond() function sends a DNS response to a request, including all of the RRs that you attached to it, with the error code err. If you get a request that you don’t want to respond to, you can ignore it by calling evdns\_server\_request\_drop() on it to release all the associated memory and bookkeeping structures.

Interface

#define EVDNS\_FLAGS\_AA 0x400

#define EVDNS\_FLAGS\_RD 0x080

**void** evdns\_server\_request\_set\_flags(**struct** evdns\_server\_request\*req, **int** flags);

If you want to set any flags on your response message, you can call this function at any time before you send the response.

All the functions in this section were introduced in Libevent 1.3, except for evdns\_server\_request\_set\_flags() which first appeared in Libevent 2.0.1-alpha.

**DNS Server example**

Example: A trivial DNS responder

#include <event2/dns.h>

#include <event2/dns\_struct.h>

#include <event2/util.h>

#include <event2/event.h>

#include <sys/socket.h>

#include <stdio.h>

#include <string.h>

#include <assert.h>

/\* Let’s try binding to 5353. Port 53 is more traditional, but on most operating systems it requires root privileges. \*/

#define LISTEN\_PORT 5353

|  |  |  |  |
| --- | --- | --- | --- |
| #define | LOCALHOST\_IPV4\_ARPA | "1.0.0.127.in-addr.arpa" |  |
| #define | LOCALHOST\_IPV6\_ARPA | ("1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0." | \ |
|  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Fast portable non-blocking network programming with Libevent | | | | | | | 118 / 120 | |  |
|  |  |  |  | | | |  |  |  |
|  |  |  | "0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.ip6.arpa") | | | |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
| **const** | ev\_uint8\_t | LOCALHOST\_IPV4[] | = | { | 127, 0, 0, 1 }; |  |  |  |
| **const** | | ev\_uint8\_t | LOCALHOST\_IPV6[] | = | { | 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,1 }; |  |  |  |

#define TTL 4242

/\* This toy DNS server callback answers requests for localhost (mapping it to 127.0.0.1 or ::1) and for 127.0.0.1 or ::1 (mapping them to localhost).

\*/

**void** server\_callback(**struct** evdns\_server\_request\*request, **void** \*data)

{

**int** i;

**int** error=DNS\_ERR\_NONE;

/\* We should try to answer all the questions. Some DNS servers don’t do

this reliably, though, so you should think hard before putting two questions in one request yourself. \*/

**for** (i=0; i < request->nquestions; ++i) {

**const struct** evdns\_server\_question\*q = request->questions[i]; **int** ok=-1;

/\* We don’t use regular strcasecmp here, since we want a locale-independent comparison. \*/

**if** (0 == evutil\_ascii\_strcasecmp(q->name, "localhost")) { **if** (q->type == EVDNS\_TYPE\_A)

ok = evdns\_server\_request\_add\_a\_reply(

request, q->name, 1, LOCALHOST\_IPV4, TTL);

**else if** (q->type == EVDNS\_TYPE\_AAAA)

ok = evdns\_server\_request\_add\_aaaa\_reply( request, q->name, 1, LOCALHOST\_IPV6, TTL);

} **else if** (0 == evutil\_ascii\_strcasecmp(q->name, LOCALHOST\_IPV4\_ARPA)) { **if** (q->type == EVDNS\_TYPE\_PTR)

ok = evdns\_server\_request\_add\_ptr\_reply(

request, NULL, q->name, "LOCALHOST", TTL);

} **else if** (0 == evutil\_ascii\_strcasecmp(q->name, LOCALHOST\_IPV6\_ARPA)) { **if** (q->type == EVDNS\_TYPE\_PTR)

ok = evdns\_server\_request\_add\_ptr\_reply(

request, NULL, q->name, "LOCALHOST", TTL);

} **else** {

error = DNS\_ERR\_NOTEXIST;

}

**if** (ok<0 && error==DNS\_ERR\_NONE)

error = DNS\_ERR\_SERVERFAILED;

}

/\* Now send the reply. \*/

evdns\_server\_request\_respond(request, error);

}

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

{

**struct** event\_base\*base;

**struct** evdns\_server\_port\*server;

evutil\_socket\_t server\_fd;

**struct** sockaddr\_in listenaddr;

base = event\_base\_new();

**if** (!base)

**return** 1;

server\_fd = socket(AF\_INET, SOCK\_DGRAM, 0);

**if** (server\_fd < 0)

**return** 2;

memset(&listenaddr, 0, **sizeof**(listenaddr));

|  |  |
| --- | --- |
| Fast portable non-blocking network programming with Libevent | 119 / 120 |

listenaddr.sin\_family = AF\_INET;

listenaddr.sin\_port = htons(LISTEN\_PORT);

listenaddr.sin\_addr.s\_addr = INADDR\_ANY;

**if** (bind(server\_fd, (**struct** sockaddr\*)&listenaddr, **sizeof**(listenaddr))<0)

**return** 3;

/\*The server will hijack the event loop after receiving the first request if the socket - is blocking\*/

**if**(evutil\_make\_socket\_nonblocking(server\_fd)<0)

**return** 4;

server = evdns\_add\_server\_port\_with\_base(base, server\_fd, 0, server\_callback, NULL);

event\_base\_dispatch(base);

evdns\_close\_server\_port(server);

event\_base\_free(base);

**return** 0;

}

**Obsolete DNS interfaces**

Obsolete Interfaces

void evdns\_base\_search\_ndots\_set(struct evdns\_base \*base, const int ndots);

int evdns\_base\_nameserver\_add(struct evdns\_base \*base, unsigned long int address);

void evdns\_set\_random\_bytes\_fn(void (\*fn)(char \*, size\_t));

struct evdns\_server\_port \*evdns\_add\_server\_port(evutil\_socket\_t socket, int flags, evdns\_request\_callback\_fn\_type callback, void \*user\_data);

Calling evdns\_base\_search\_ndots\_set() is equivalent to using evdns\_base\_set\_option() with the "ndots" option.

The evdns\_base\_nameserver\_add() function behaves as evdns\_base\_nameserver\_ip\_add(), except it can only add nameservers with IPv4 addresses. It takes them, idiosyncratically, as four bytes in network order.

Before Libevent 2.0.1-alpha, there was no way to specify a event base for a DNS server port. You had to use evdns\_add\_server\_port() instead, which took the default event\_base.

From Libevent 2.0.1-alpha through 2.0.3-alpha, you could use evdns\_set\_random\_bytes\_fn to specify a function to use for generating random numbers instead of evdns\_set\_transaction\_id\_fn. It no longer has any effect, now that Libevent provides its own secure RNG.

The DNS\_QUERY\_NO\_SEARCH flag has also been called DNS\_NO\_SEARCH.

Before Libevent 2.0.1-alpha, there was no separate notion of an evdns\_base: all information in the evdns subsystem was stored globally, and the functions that manipulated it took no evdns\_base as an argument. They are all now deprecated, and declared only in event2/dns\_compat.h. They are implemented via a single global evdns\_base; you can access this base by calling the evdns\_get\_global\_base() function introduced in Libevent 2.0.3-alpha.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Current function | Obsolete global-evdns\_base version |  |
|  | event\_base\_new() | evdns\_init() |  |
|  | evdns\_base\_free() | evdns\_shutdown() |  |
|  | evdns\_base\_nameserver\_add() | evdns\_nameserver\_add() |  |
|  | evdns\_base\_count\_nameservers() | evdns\_count\_nameservers() |  |
|  | evdns\_base\_clear\_nameservers\_and\_suspend() | evdns\_clear\_nameservers\_and\_suspend() |  |
|  | evdns\_base\_resume() | evdns\_resume() |  |
|  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Fast portable non-blocking network programming with Libevent | | | 120 / 120 | |  |
|  |  |  |  |  |  |
|  | Current function |  | Obsolete global-evdns\_base version |  |
|  | evdns\_base\_nameserver\_ip\_add() |  | evdns\_nameserver\_ip\_add() |  |  |
|  | evdns\_base\_resolve\_ipv4() |  | evdns\_resolve\_ipv4() |  |  |
|  | evdns\_base\_resolve\_ipv6() |  | evdns\_resolve\_ipv6() |  |  |
|  | evdns\_base\_resolve\_reverse() |  | evdns\_resolve\_reverse() |  |  |
|  | evdns\_base\_resolve\_reverse\_ipv6() |  | evdns\_resolve\_reverse\_ipv6() |  |  |
|  | evdns\_base\_set\_option() |  | evdns\_set\_option() |  |  |
|  | evdns\_base\_resolv\_conf\_parse() |  | evdns\_resolv\_conf\_parse() |  |  |
|  | evdns\_base\_search\_clear() |  | evdns\_search\_clear() |  |  |
|  | evdns\_base\_search\_add() |  | evdns\_search\_add() |  |  |
|  | evdns\_base\_search\_ndots\_set() |  | evdns\_search\_ndots\_set() |  |  |
|  | evdns\_base\_config\_windows\_nameservers() |  | evdns\_config\_windows\_nameservers() |  |  |

The EVDNS\_CONFIG\_WINDOWS\_NAMESERVERS\_IMPLEMENTED macro is defined if and only if evdns\_config\_windows\_name is available.