VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DEVELOPER SUPPORT TOOLS FOR TEVENT LIBRARY

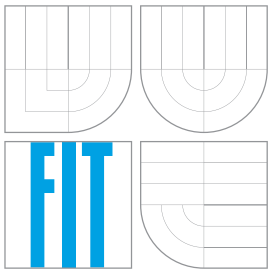
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DAVID KOŇAŘ

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VÝVOJÁŘSKÉ NÁSTROJE KE KNIHOVNĚ TEVENT

DEVELOPER SUPPORT TOOLS FOR TEVENT LIBRARY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — DAVID KOŇAŘ
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN ZELENÝ
SUPERVISOR

BRNO 2013

Abstrakt

Práce se zabývá vytvořením návodu pro knihovnou tevent. Přiblížena je samotná koncepce knihovny a její možnosti spolu s ukázkami kódu, jak s knihovnou vhodně pracovat. Dále se práce zabývá rozšířením pro debuggery, jež bylo současně s touto prací vytvořeno a které umožňuje efektivnější práci s touto knihovnou. Zahrnuto je rovněž porovnání s konkurující knihovnou libevent.

Abstract

Aim of this thesis is creation of description and tutorial for tevent library. Another goal was developing of debugger extension which has been created along with this thesis and is helpful for programmers working with tevent. Furthermore, there is a comparison of tevent with competition library - libevent.

Klíčová slova

tevent, událostmi řízené programování, událost, upozornění, asynchronní programování.

Keywords

tevent, event-based, event-driven, event, notification asynchronous programming.

Citace

David Koňař: Developer Support Tools for tevent Library, bakalářská práce, Brno, FIT VUT v Brně, 2013

Developer Support Tools for tevent Library

Statement

I hereby declare that this thesis is my own work and effort and it has not been previously submitted for any degree. Where other sources of information have been used, they have been acknowledged. This thesis was elaborated under the supervision of Ing. Jan Zelený.

.....
David Koňař
May 15, 2013

Acknowledgements

I would like to thank Ing. Jan Zelený for his guidance and mentoring. I would further like to thank people from Red Hat for feedback.

Contents

Chapter 1

Introduction

The efficient use of processing time without any unnecessary waiting was, is and will continue to be one of programmers' most basic goals. Performing an operation while others, previously computed, are waiting for its inputs is the key to more economical and efficient applications. Asynchronous or so-called nonblocking programming allows other processing to continue before an awaiting event occurs [8]. This attitude permits the execution of other operations instead of inefficient waiting time. Such behaviour can be very useful especially when dealing with events where time delay or waiting is expected (e.g. internet communication, user input, etc.).

The event-based programming model represents programs driven by events which are subsequentially handled according to the programmer's will. Such behavior is nowadays very common and widely used, especially in GUI implementations and applications which provide communication over the internet. While waiting for a specific event to happen, the program runs other instructions - provides different tasks in general - and at the time when the awaited event happens, then deals with it according to set orders. In this area such a handler is usually called a *callback* function which is triggered and its purpose is to deal with the occurred event. The callback function is set together with a request to wait for the specific event so that a program immediately knows what to do as soon as the requested action is noticed.

Tevent[1] belongs to group of event-based libraries (libevent, libev, etc.) which deal with asynchronous tasks. The variety of events supported by tevent library is quite extensive (e.g. timer, signals) although in comparison to other similar libraries tevent still has potential for further development. The differences between these libraries are mainly in their speed of preparing handlers and processing events, and also in the range or number of "backends" (polling calls) which are supported.

Tevent is not widely spread among projects, but it is used in the Samba project[2]. So far the documentation for it is quite restricted, which limits the use of the library by others than the developers of Samba. I found it useful and interesting to contribute to the improvement of the documentation and tutorial for the tevent library, whose use in such a big project could be starting point for its further use elsewhere.

This thesis consists of five chapters and two appendices. The concept of event programming is described in the following chapter, together with libraries that provide API for working with events. The tevent library itself is introduced at the beginning of chapter

[1] http://tevent.samba.org/
[2] http://samba.org

3. Furthermore, chapters 3.2, 3.3 and 3.4 take a close look at the main features of tevent library - tevent context, event handling and asynchronous requests. All three chapters are divided into several sections, dealing with individual questions and offering practical examples of tevent usage in development.

Chapter 4 focuses on a plugin that has been created together with this thesis. The chapter includes the plugin's installation and configuration, and examples of its usage. The thesis is summarized and concluded in Chapter 5. Benchmark testing between tevent and other libraries is discussed in the final Appendix B.

Chapter 2

About the Event-driven Programming Paradigm

Event-based programming is a name for a programming paradigm that presents a different attitude to program flow and input data. Programming paradigms such as imperative, object-oriented, logic or functional are grounded on the fundamental idea that progress within a program is predetermined, whereas programs created in the event-based programming paradigm do not predict in advance the sequences that will occur. They are written to react to events. These events (input data) in fact determine what particular task will be performed by the program [16].

This figure shows a pattern for handling events, demonstrating the concept of working with events.

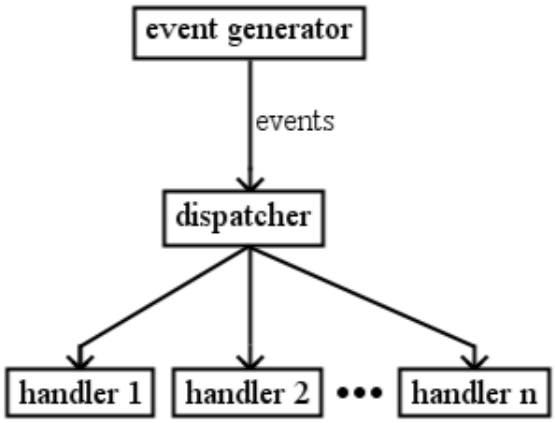


Figure 2.1: Handling of events, [6]

The basic concept behind the behaviour of event-based programs can be split into just 2 parts, where both of them are essential and the first is precondition for the second.

1. event loop - waiting for and detecting events
2. event handling

The reason for using this paradigm for software development rather than another may be seen in the possibilities which it offers. If we compare software using threads to programs based on events, there are several considerations to bear in mind.

Although threads can offer the opportunity to write a code which retains the appearance of serial programming while having the ability to utilize multiprocessor hardware, the fact of having to deal with shared resources that need to be protected cautiously might be a problem. As the complexity and robustness of the application rises, more and more requirements relating to shared data and the conditions coordinating the execution of thread must be dealt with, somehow. This is very demanding and leads to bugs in the source code. Programming with events offers a means of preventing bugs like these that would result from dealing with concurrency and synchronization between threads. Event-driven programming is typical for numerous smaller callback functions with a grounding in dynamic memory allocation [4].

The reason why this programming paradigm should be used is that it provides the possibility of creating more interactive applications (nowadays widely used for GUI programs) or less bug-prone programs, in contrast to multiprocessor programs. In fact, it is possible to extend event-driven programming with the ability to utilize the advantages of multiprocessor hardware. By *colouring* events it is possible to recognize what can be handled in parallel and what cannot [4] [17].

On the other hand, the implementation of this event-driven principle might lead to unclear code. The source code can seem to be less obvious to the programmer because of the unpredictability of the program flow and the fact that the programmer's habitual attitudes to programming are more likely to be imperative than event-driven.

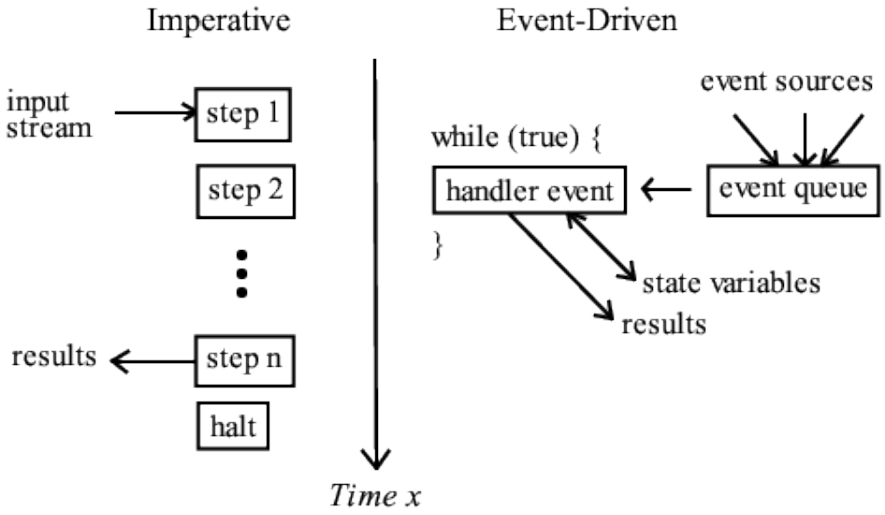


Figure 2.2: Imperative and Event-Driven paradigm contrasted, [16]

In this thesis, the term *event* is used very often and it is a key term. By this term *event* we mean something interesting - an action whose origin is very often outside of the program itself, although it may also be created by another part of the program. The means of handling the occurred event is also called the *service*. Events detected by the program, which then triggers the associated service are, for example [5]:

- user input
- signals
- time events

- file descriptor events

In order to handle these events more easily, there are several libraries that provide abstraction over low-level programming and come with API for working in an event-driven style. So far, there are not many of these libraries and their development is still very much an active project.

2.1 Libraries for Working with Events

There are several libraries that provide API for catching and servicing events, which differ in the number of developers maintaining their codes, the platforms they support, and their performance.

Libevent

This is a library offering API for dealing with a wide range of events, a feature common to all of these libraries, and additionally libevent contains a framework for buffered network IO, and support for several protocols, such as DNS, HTTP, and others. It is widely used in applications such as Chromium, Tor, and Crawl, and the library is still in development [13]. In comparison to other libraries, it offers broad support of different platforms. There is a tutorial on the library website describing how to work with this library.

The project's homepage is at http://libevent.org/.

Libev

A younger library, libev is loosely inspired by libevent. Like libevent, it supports a wide range of events but also tries to offer additional possibilities (e.g. PID[1] watchers) and more efficient processing. The efficiency of both libraries was last tested by a libev developer in 2011, and the results showed higher performance from libev [10] [9].

The library's website is located at http://software.schmorp.de/pkg/libev.html.

Liboop

Liboop is another library that provides a notification interface. It does not provide so broad range of features and its supported backends are just inefficient system calls *select* and *poll*. The last stable version was released in 2003. Although this library has not been updated or developed for a long time, it is still present in e.g. current Ubuntu distributions where other applications such as *libruli* are built on the basis of this event library.

The homepage of the library is at http://directory.fsf.org/wiki/Liboop.

Tevent

This library, which is still in development, offers all the main characteristic features of this type of library. In comparison with libev or libevent it is clearly not so extensive and does not offer so many features, however its great benefit is its memory management by means of the talloc library, and the possibility of creating of nested requests.

The library's homepage is located at http://tevent.samba.org/.

[1] process ID

Comparison of Libraries

This brief table shows a comparison of the libraries (a deeper and more thorough look at tevent library in contrast to libevent can be found in Appendix B).

	System calls	Platform	Release date	Notes
libev 4.15	epoll, poll, kqueue, select, event ports	Linux, *BSD, Mac OS X, Solaris, Windows	01 March, 2013	More efficient than libevent
libevent 2.0.21	poll, kqueue, event ports, epoll, select, Windows select	Linux, *BSD, Mac OS X, Solaris, Windows [a]	18 November, 2012	Well-known for a long time and widely used in projects
liboop 1.0	select, poll	Linux, *BSD, Mac OS X, Windows	27 October, 2003	Not being developed
tevent 0.9.17	select, poll, epoll	Linux, *BSD, Mac OS X, Windows	17 August, 2012	Support of kqueue is in progress

[a] libevent offers additional features because it supports Input/Output Completion Port (IOCP)

Table 2.1: Comparison of event notification libraries

There is also a further library, libverto, which operates abstractly above the abovementioned libraries and offers a neutral event loop API for asynchronous programming interface. Programmers using this do not have to deal with specific libraries individually, but can use unified calls and choose which library should be used specifically in the background.

2.2 Polling Mechanisms

A very significant influence on the efficiency of each library lies in the polling mechanisms that the library supports on individual operating systems. As there are many operating systems, it is not unexpected that several mechanisms of dealing with events have been created. Together with a time factor that leads to the development of more efficient system calls providing communication between the application and the operating system, there are now also several of these mechanisms, each of them supported by different platforms. This fact places higher demands on the cross-platform portability of event libraries while maintaining efficient performance.

Tevent library supports several mechanisms for dealing with events, specifically: *select*, *poll* and *epoll* (*kqueue*[2] support is currently being discussed and is supported in upstream, but no stable version with this feature has yet been introduced). In contrast to its competitors, such as libevent or libev, tevent does not so far have the capacity to work with the

[2] http://www.freebsd.org/cgi/man.cgi?query=kqueue

event ports[3] framework for Solaris nor with the *kqueue* mechanism which was introduced in the FreeBSD operating system. Both *epoll* and *kqueue* provide a much more efficient method of event notification in comparison with *select* or *poll*.

The system calls *epoll* and *kqueue* operate at O(1) instead of the inefficient *select* and *poll* which cause O(n) performance. The complexity of these mechanisms is much more evident when working with a large number of events [11] [12]. Although both of these possibilities represent great advancement in performance, there are distinguishing features which make their use appropriate in different cases. The advantages of *kqueue* compared to *epoll*, disregarding a slight performance difference[4] in favour of kqueue[13], is its more abstract point of view, which leads to it supporting various events connected with e.g. signals or processes. The *epoll* mechanism is limited in this respect and it is not capable of e.g. handling files stored on a disc by using one unified system call [7].

[3] http://dsc.sun.com/solaris/articles/event_completion.html
[4] This benchmark was run with libevent library.

Chapter 3

Tevent

Tevent is a library that simplifies work with events and is based on the talloc memory management library[15]. It offers an API providing not just individual event requests but also complex nested series of interconnected asynchronous requests. It allows the user to set freely as many time events as required, handlers for reading and writing per file descriptor, or handlers which take care of signals. The main feature of this library is the creation of asynchronous requests that may be nested in others. In case of need, tevent also provides its own FIFO[1] queue, which solves the problem of needing to carry out specific operations sequentially.

For tevent's memory needs, the talloc library is used and therefore the allocated data are stored in a hierarchical structure that allows both easy management and the freeing of used memory. All the other source code is created on the basis of the standard ANSI C libraries.

3.1 Basic Concepts of tevent

Tevent library is built on a common concept for event-driven programming corresponding to the scheme shown in Figure 2.1. Each of the important parts of tevent library is described in the following chapters.

An essential element of the tevent library is its internal structure called *tevent context* (Section 3.2), which represents a root unit. At least one such context must be created, and all events have to belong under a context.

The principle of callback execution consists of a loop that awaits events and after noticing them (Section 3.3), hands them over to a callback function. According to this process, at least one event has to be registered before the loop is started, otherwise the loop would not be able to catch any event. Setting up other event handlers may, however, be done on the run and nested in other handlers (there is an example that demonstrates this possibility on the attached CD).

The most specific and widely used part of this library is asynchronous computation, which, if we tear the code apart, is based on events. However, the API that tevent has come up with also allows us to create much more complex mechanisms (Section 3.4). The asynchronous computation in this library is composed of so-called *tevent requests* which may be set either individually or in a hierarchial structure (the term *subrequest* will be further used for these nested *tevent requests*).

[1] First In, First Out

The flow of the program depends on the success or failure of the requests and all the callbacks and superior requests react based on this outcome.

Memory Management with talloc Library

Tevent uses talloc library for its memory needs. Deep knowledge of this library is not necessary in order to work with tevent, but a few function calls are essential. Proper knowledge about working with talloc will also significantly ease tasks with tevent. Usage of talloc for other memory requirements as well, when working with tevent, is highly recommended because it will enable easy management and deallocation all of the used memory. The following section introduces talloc library very briefly, covering only the humblest requirements in order to understand how to work with talloc in cooperation with tevent.

In the next Section 3.2, the basic initialization of the absolutely essential (`TALLOC_CTX`) variables is shown - this is based on the function

```
void* talloc_new (const void *ctx)
```

which, in case the passed argument is `NULL`, allocates a parent (top-level) memory unit. For another allocation of memory needs, calling

```
void* talloc (const void *ctx, #type)
```

will be appropriate. The first argument represents a parent memory unit (in this case it would be the pointer returned from previous `talloc_new(...)` call) and the second is the type of requested memory, e.g. `int`, `char`, some structure, etc. The returned value is a pointer to memory of the requested size and can be treated the very same way as an output from `malloc` function.

To free allocated memory, the following function is used:

```
int talloc_free(void *ptr)
```

This will free the memory pointed out by the pointer `ptr` and all descendant units that were allocated by `talloc(...)` and the pointer in the first argument. Similarly, calling `talloc_free(root_ctx)` at the very end of the program will lead to total memory deallocation (if all of the memory needs were allocated within the program run and in accordance with talloc principles).

More information about this memory system can be found in the tutorial and documentation on talloc's website http://talloc.samba.org/.

3.2 Tevent Context

Background of tevent Context

A tevent context is this library's basic logical unit for working with events. It has to be created, initialized, and all the events have to be registered within a tevent context so they can be caught later. Of course several tevent contexts may be created individually according to the need to process different types of events at distinctive points in the program (within different tevent loops).

The following code demonstrates initialization of tevent context which is a node within a hierarchical memory tree managed by talloc. In this case a root node is allocated first, so the other memory requirements (in this case only tevent context) can later become its descendants.

Hereinafter in the examples given in this thesis such an initialization is excluded, but it is assumed to have been performed. The absence of these lines of code in any program would cause it to malfunction.

```
TALLOC_CTX *mem_ctx = talloc_new(NULL);
if(mem_ctx == NULL) {
    /* error handling */
}

struct tevent_context *ev_ctx = tevent_context_init(mem_ctx);
if(ev_ctx == NULL) {
    /* error handling */
}
```

Listing 3.1: tevent context initialization

As mentioned, tevent memory requirements are managed by talloc library which is a hierarchial, reference pool system in which, if a node of allocated memory is deallocated, all the descendants are freed altogether, automatically and with no further concern. Based on the hierarchial structure used by talloc, a root node has to be created and all other memory requests should be allocated as the root's descendants.

Using this system is very helpful and efficient because by deallocating a root node the whole tree of allocated memory is freed, thereby greatly reducing the probability of memory leak [3].

It is possible to identify several stages for the tevent context:

- **Initialization** is started by calling `tevent_context_init(...)` which launches preparations for further operations in the back end of tevent - the library deals with the allocation of memory via talloc and checks the availability of system calls: *select*, *poll* or *epoll*. *Epoll* (as the most efficient of these system calls (see Section 2.2) mechanism is usually used as a default, if it is available under current operating system).

- As soon as the first tevent context is initialized, the library is ready and it provides simple API for **setting up handlers**. It is up to the programmer to set a file descriptor, to set a time (scheduled or immediate), or to signal events. A description of the particular capabilities and examples of each type of event are given in the following chapters.

- **Removing events** from a tevent context is possible if they have not already been caught and processed. In order to remove event registration with a handler, calling simple memory deallocation of the structure representing the event in question is sufficient. Because tevent works in cooperation with talloc, the function `talloc_free(...)` (mentioned in Section 3.1) will do this job.

Structure of tevent Context

This Figure shows the structure that is created along with requests for event handling. Each type of event is stored separately in lists. In addition to the lists shown in the diagram, the tevent context also contains many other data (e.g. information about the available system mechanism for triggering callbacks) but this information is used in the library's backend and is not affected by the program itself.

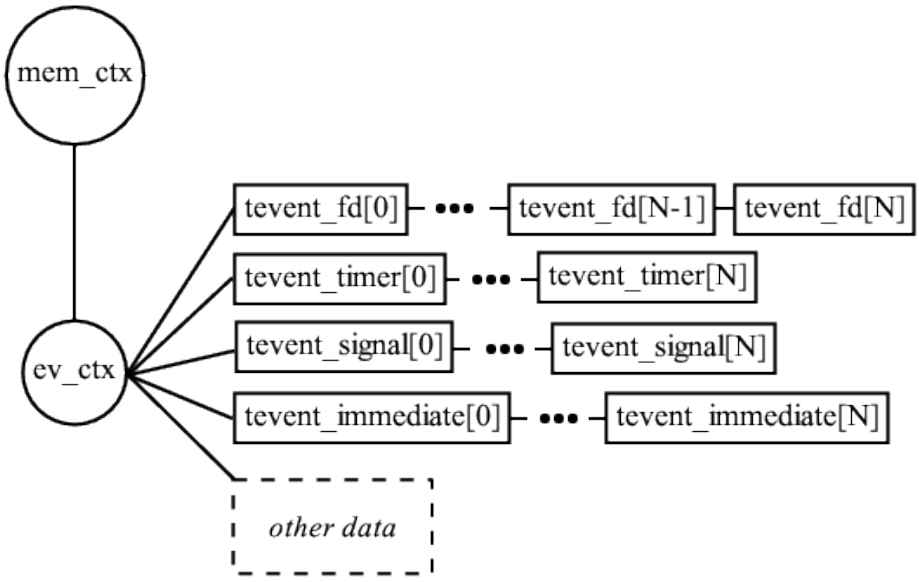


Figure 3.1: Tevent context structure

Once the tevent context has been initialized, it is then possible to register events to be captured, and handlers to covering the processing. It is important to mention that if the event were to occur at this point, the application would not catch it and therefore no callback would be invoked. The following essential step starts the tevent loop which behaves as a dispatcher (as shown in Figure 2.1).

Tevent Loops

Tevent loops are the dispatcher for events. They catch them and trigger the handlers. In the case of longer processes, the program spends most of its time at this point waiting for events, invoking handlers and waiting for another event again.

There are 2 types of loop available for use in tevent library:

- `int tevent_loop_wait (struct tevent_context *ev_ctx)`
- `int tevent_loop_once (struct tevent_context *ev_ctx)`

The only difference between these 2 functions is whether the loop will theoretically last for ever. Calling `event_loop_wait(...)` will set up waiting that will continue for as long as there is an event registered in tevent context. It can be interrupted only by the application crashing, by sending the appropriate signal to the program, or by internal termination of the program. By contrast, calling `event_wait_once(...)` will allow just one loop, the

first event, complete, and then the loop will break. Examples showing the difference can be found on the attached CD.

These functions take as their only argument the pointer to a structure `tevent_context`, so only those events registered within the specific context which is set in the specific loop will be caught. Therefore, if there is no reason to distinguish various events into different contexts, it is sufficient simply to place all the events under a single context.

The returning values indicate whether the loop was succesful (*zero* value), or not (*nonzero* value).

3.3 Tevent - Managing Events

In the following subsections, the ability to process various types of event with callbacks is introduced in more detail. Before this, however, it is advisable to be familiar with several *typedefs* which are laid down by the library. These newly set functions are provided by tevent in order to pass on the most important data about the relevant event in a straightforward manner.

In tevent library, the important data which connect nodes in the hierarchical memory system and links both required and optional information, are the aforementioned structures, `TALLOC_CTX` and `tevent_context`. Each of these functions features a callback for specific event and includes, in addition, further arguments which allow more precise work with the event. The most significant argument, included in each of these functions, is the `void` pointer to data, which permits the programmer to send any kind of data into the event handler. These data are called *private data*.

```
typedef void(*tevent_fd_handler_t)(struct tevent_context *ev,
                                   struct tevent_fd *fde, uint16_t
                                   flags, void *private_data)


typedef void(*tevent_timer_handler_t)(struct tevent_context *ev,
                                      struct tevent_timer *te,
                                      struct timeval
                                      current_time, void
                                      *private_data)


typedef void(*tevent_immediate_handler_t)(struct tevent_context
                                          *ev, struct
                                          *tevent_immediate *im,
                                          void *private_data)


typedef void(*tevent_signal_handler_t)(struct tevent_context *ev,
                                       struct tevent_signal *se,
                                       int signum, int count,
                                       void *siginfo, void
                                       *private_data)
```

The names of the functions themselves clearly describe to what particular event each of the functions belongs. All of them are presented and described more in detail in the next part of this thesis, where each of these *typedefes* corresponds to the focus and title of each subsection.

3.3.1 File Descriptor Event

Support of events on file descriptors is mainly useful for socket communication but it certainly works flawlessly with standard streams (`stdin`, `stdout`, `stderr`) as well. Working asynchronously with file descriptors enables switching within processing I/O operations. This ability may rise with a greater number of I/O operations and such overlapping leads to enhancement of the throughput.

This is our first meeting with setting event handlers, so let us first show how to register a handler for an event, and then introduce the arguments of the function further.

```
struct tevent_fd* tevent_add_fd (struct tevent_context *ev,
                                 TALLOC_CTX *mem_ctx, int fd,
                                 uint16_t flags,
                                 tevent_fd_handler_t handler,
                                 void *private_data)
```

The first two arguments of this function have already been introduced in this thesis and for now, the most important thing we must note about them is that both must not be `NULL`. The first, *tevent_context*, is a reference to the parent element under which the file descriptor event will be registered. This is important because the tevent loop takes as its only argument the pointer to *tevent_context* and only an event within this will be caught and handled.

The second one is a memory pointer representing a node allocated within the hierarchical talloc memory system. In terms of memory allocation, the future file descriptor event will be a direct descendant of `mem_ctx` (the freeing of `mem_ctx` will result in the freeing of `struct tevent_fd*` too).

The third argument is an integer value representing the opened file descriptor[2].

The unique argument for the `tevent_add_fd(...)` call is the fourth one - this is not used in any other function managing events. The flag defines the type of event at the file descriptor which we want to be notified about. It can be either reading or writing - analogous macros are defined in the tevent library for usage: `TEVENT_FD_READ` and `TEVENT_FD_WRITE`.

The fifth argument is similar in every function, but differs in type. This argument represents the specific handler that will be triggered as soon as the file descriptor event occurs.

The last, sixth, argument is common for every function that registers an event. Theoretically it can be `NULL`, in case that event processing does not request to hand over any data related to the event. Otherwise, this pointer refers to a variable (integer, array, structure, etc.) that keeps data necessary for processing the event, storing temporary data, etc.

An example of establishing a new event handler for a file descriptor where reading data is demanded could look like this:

[2] As a precaution, it is essential that this argument is a file descriptor and not a file pointer

```
int run(TALLOC_CTX *mem_ctx, struct tevent_context *event_ctx) {
   struct tevent_fd* fd_event = NULL;

   fd_event = tevent_add_fd (event_ctx, mem_ctx, fd,
                             TEVENT_FD_READ, handler, buffer);
   if(fd_event == NULL) {
      /* error handling */
   }
   return tevent_loop_once();
}
```

Listing 3.2: Capturing file descriptor event

This code snippet describes the simplest usage of tevent library, which will execute a callback handler as soon as some data is readable on a specific file descriptor (for example an opened socket through the internet). To guarantee the feasibility of this code it is essential to check the memory allocation for errors. At the moment when any part of the whole fails, appropriate actions must be taken.

Of course these lines of code may be placed within a much more complex asynchronous tevent request, as I describe more precisely in Chapter 3.4).

There are several other functions included in tevent API related to handling file descriptors (there are too many functions defined within tevent therefore just some of them are fully described within this thesis. The declaration of the rest can be easily found on the library's website or directly from the source code):

- `tevent_fd_set_close_fn(...)` - can add another function to be called at the moment when a structure `tevent_fd` is freed.
- `tevent_fd_set_auto_close(...)` - calling this function can simplify the maintenance of file descriptors, because it instructs tevent to close the appropriate file descriptor when the `tevent_fd` structure is about to be freed.
- `tevent_fd_get_flags(...)` - returns flags which are set on the file descriptor connected with this tevent_fd structure.
- `tevent_fd_set_flags(...)` - sets specified flags on the event's file descriptor.

As mentioned, a more elaborate example using the potential possibilities of tevent is shown below.

```
static void close_fd(struct tevent_context *ev, struct tevent_fd
                     *fd_event, int fd, void *private_data) {
  /* processing when fd_event is freed */
}

static void handler(struct tevent_context *ev, struct tevent_fd
                    *fde, uint16_t flags, void *private_data) {
  /* handling event; reading from a file descriptor */
  tevent_fd_set_close_fn (fd_event, close_fd);
}
```

```
int run(TALLOC_CTX *mem_ctx, struct tevent_context *event_ctx,
        int fd, uint16_t flags) {
   struct tevent_fd* fd_event = NULL;

   if(flags & TEVENT_FD_READ) {
      fd_event = tevent_add_fd (event_ctx, mem_ctx, fd, flags,
                                handler, buffer);
   }
   if(fd_event == NULL) {
      /* error handling */
   }
   return tevent_loop_once();
}
```

Listing 3.3: More complex example of `tevent_add_fd()` usage

This example register a handler for an event on file descriptor when flag is set for reading.

3.3.2 Time Event

Working with timed events is similar to working with file descriptor events. Timed events are used when triggering a callback is required at a specific time.

Time events differ in the argument that specifies the time when the callback should be invoked. The time value should be in the future, or at the current time. If a function needs to be triggered at the very moment, it is worth to considering whether usage of *immediate event* may be more suitable (see Section 3.3.4).

```
struct tevent_timer* tevent_add_timer (struct tevent_context *ev,
                                       TALLOC_CTX *mem_ctx, struct
                                       timeval next_event,
                                       tevent_timer_handler_t
                                       handler, void
                                       *private_data)
```

Tevent also defines a few more functions that help when working with timed actions. The complete list of these is included in tevent's online documentation[3]. In this thesis, only a few of these are mentioned, described and demonstrated with examples of their usage.

Returns standard timeval[4] structure containing time value of current time.

```
struct timeval tevent_timeval_current (void)
```

Returns time value in the future created by adding specified offset to current time.

```
struct timeval tevent_timeval_current_ofs (uint32_t secs,
                                           uint32_t usecs)
```

Returns time value resulting from combination of specified timeval structure and amount of time.

[3] http://tevent.samba.org/

[4] http://pubs.opengroup.org/onlinepubs/000095399/basedefs/sys/time.h.html

```
struct timeval tevent_timeval_add (const struct timeval *tv,
                                   uint32_t secs, uint32_t usecs)
```

Based on this, a more sophisticated example showing a cyclic event follows. The example repeatedly triggers the handler with 2 seconds delay over one minute (callback will be invoked thirty times).

```
struct foo_state {
   struct timeval endtime;
   struct TALLOC_CTX *ctx;
};

static void foo(struct tevent_context *ev, struct tevent_timer
                *tim, struct timeval current_time, void
                *private_data) {

   struct foo_state *data = talloc_get_type(private_data, struct
                                            foo_state);

   struct tevent_timer *time_event;
   struct timeval schedule;
   if(tevent_timeval_compare(&current_time, &(data->endtime)) <
      0) {

        schedule = tevent_timeval_current_ofs(2, 0);
        time_event = tevent_add_timer(ev, data->ctx, schedule,
                                      foo, data);
     }
}

int run(TALLOC_CTX *mem_ctx, struct tevent_context *event_ctx) {

   schedule = tevent_timeval_current_ofs(2, 0);
   foo->endtime = tevent_timeval_add(schedule, 60, 0);
   foo->ctx = mem_ctx;

   time_event = tevent_add_timer(event_ctx, mem_ctx, schedule,
                                 foo);
   if(time_event == NULL) {
      /* error handling */
   }
   return tevent_loop_wait();
}
```

Listing 3.4: Complex example of `tevent_add_timer()` usage

3.3.3 Signal Event

Another feature that tevent offers catching and handling signals. This is an alternative to standard C library functions `signal()` or `sigaction()`. The main difference that distinguishes these ways of treating signals is their setting up of handlers for different time intervals of the running program.

While standard C library methods for dealing with signals offer sufficient tools for most cases, they are inadequate for handling signals within the tevent loop. It could be necessary to finish certain tevent requests within the tevent loop without interruption. If a signal was sent to a program at a moment when the tevent loop is in progress, a standard signal handler would not return processing to the application at the very same place and it would quit the tevent loop for ever. In such cases, tevent signal handlers offer the possibility of dealing with these signals by masking them from the rest of application and not quitting the loop, so the other events can still be processed.

Calling the function `tevent_add_signal(...)` sets up a callback for the given signal and it is quite similar to the signal events.

```
struct tevent_signal *tevent_add_signal (struct tevent_context *ev,
                                         TALLOC_CTX *mem_ctx,
                                         int signum, int sa_flags,
                                         tevent_signal_handler_t
                                         handler,
                                         void *private_data)
```

In this case, the third and fourth arguments are unique to this function. Integer `signum` is the number of the signal (`SIGINT`, `SIGCHILD`, etc.) and integer `sa_flags` represents a specification for dealing with the caught signal. The values correspond with standard `signaction()`[5].

A control function, which enables us to verify whether it is possible to handle signals via tevent, is defined within tevent library and it returns a boolean value revealing the result of the verification.

```
bool tevent_signal_support (struct tevent_context *ev)
```

Checking for signal support is not necessary, but if it is not guaranteed, this is a good and easy control to prevent unexpected behaviour or failure of the program occurring. Such a test of course does not have to be run every single time you wish to create a signal handler, but simply at the beginning - during the initialization procedures of the program. After that, simply adapt to each situation that arises.

```
static void handler (struct tevent_context *ev,
                     struct tevent_signal *se, int signum,
                     int count, void *siginfo,
                     void *private_data) {

   /* processing event */
}
```

[5] http://linux.die.net/man/2/sigaction

```
int run(TALLOC_CTX *mem_ctx, struct tevent_context *event_ctx) {

   if(tevent_signal_support(event_ctx)) {
      tevent_sig = tevent_add_signal (event_ctx, mem_ctx,
                                      SIGINT, 0, handler, NULL);

      if(tevent_sig == NULL) {
           /* error handling */
      }
   } else {
      /* alternative signal handling */
   }
   return tevent_loop_once();
}
```

Listing 3.5: Complex example of handling signals

3.3.4 Immediate Event

These events are, as their name indicates, activated and performed immediately. It means that this kind of events have priority over others (except signal events). So if there is a bulk of events registered and after that a tevent loop is launched, then all the immediate events will be triggered before the other events. This also implies that if such an immediate event occurs within another's event handler, this immediate event will be triggered with priority over all the others events (if any) registered in the *tevent_contex* except other immediate events and signal events.

Immediate events, according to the diagram of tevent context - Figure 3.1 on page 12, are stored in a queue and processed sequentially. Therefore the expression *immediate* may not correspond exactly to the dictionary definition of *something without delay*[6] but rather *as soon as possible* after all preceding immediate events.

In order to establish a new immediate event, 2 functions have to be called, where the first one must not return `NULL`. The return value of the first function is passed as the first argument of the second function.

```
struct tevent_immediate *tevent_create_immediate (TALLOC_CTX
                                                  *mem_ctx)


void tevent_schedule_immediate (struct tevent_immediate *im,
                                struct tevent_context *ctx,
                                tevent_immediate_handler_t handler,
                                void *private_data)
```

This table clearly shows the priority between different types of events so as to clarify the order of their handling.

[6] according Longman Dictionary of Contemporary English 5th Edition

Time	ID	Function	Note
	1	`tevent_create_immediate()`	
⋮	2	`tevent_add_fd()`	
	3	`tevent_create_immediate()`	
x	4	`tevent_add_timer()`	sets event for current time x
x+k	5	`tevent_create_immediate()`	
x+l	-	`tevent_loop_wait()`	
x+m	1	immediate event	
x+n	3	immediate event	
	5	immediate event	
⋮	4	time event	
	2	file descriptor event	

Table 3.1: Priority of handling different types of events.

To prevent confusion between functions, each of the events created was given a unique ID in the table so that the steps taken after the tevent loop in processing each event is evident.

If a signal was caught at any time after the tevent loop had started it would be handled as the next event.

3.4 Asynchronous Computation with tevent

A specific feature of the library is the tevent request API that provides for asynchronous computation and allows much more interconnected working and cooperation among functions and events. When working with tevent request it is possible to nest one event under another and handle them bit by bit. This enables the creation of sequences of steps, and provides an opportunity to prepare for all problems which may unexpectedly happen within the different phases. One way or another, subrequests split bigger tasks into smaller ones which allow a clearer view of each task as a whole.

Understanding this part of the tevent library is somewhat more demanding to begin with than the material covered in the previous chapters concerning hanging a callback for a specific event. Dealing with tevent request structures and functions leads to much more puzzling behaviour in the program. It is not possible to look at code in the same way as we would in an imperative paradigm where the flow is sequential and quite easily predictable.

Because of the minimal usage of this library among developers, there is so far only one convention for writing source code within this library. This convention is based on specific naming of functions, with each containing code relating to its naming. Special naming includes not just titles for functions but also for the *private data* structures (usually) which are used for the storage of information required for each of the subrequests. The naming is based upon suffixes which differ for each of the functions, while the rest of the name is identical for all.

This thesis maintains these customs and an example of naming is shown here with the exemplary function `foo()` (further in this Chapter the creation of nested requests is described). It is possible to distinguish functions and variables based on time when they are performed:

- Functions triggered before the event happens. These establish a request.
  - `foo_send(...)` - this function is called first and it includes the creation of a tevent request - *tevent_req* structure (described in Section 3.4.1). It does not block anything, it simply creates a request, sets a callback (`foo_done`) and lets the program continue
- Functions as a result of event.
  - `foo_done(...)` - this function contains code providing for handling itself and based upon its results, the request is set either as a done or, if an error occurs, the request is set as a failure (see Section 3.4.4).
  - `foo_recv(...)` - this function contains code which should, if demanded, access the result data and make them further visible. The `foo_state` should be deallocated from memory when the request's processing is over and therefore all computed data up to this point would be lost. The principle for accessing data stored within `tevent_req` structures or as *private data* for callbacks is presented in Section 3.4.3.

As was already mentioned, specific naming subsumes not only functions but also the data themselves:

- `foo_state` - this is a structure. It contains all the data necessary for the asynchronous task.

Naming functions according to this pattern is not obligatory but is highly recommended and it is considered good practice. The structure and lucidity of the source code become much more evident when this naming is introduced, and it helps with maintenance or future modification when it is clear that e.g. *stealing of context*[7] or handing over some data to another location has been carried out within one function.

3.4.1 Creating a New Asynchronous Request

The first step for working asynchronously is the allocation of memory requirements. As in previous cases, the talloc context is required, upon which the asynchronous request will be tied. The next step is the creation of the request itself.

```
struct tevent_req* tevent_req_create (TALLOC_CTX *mem_ctx,
                                      void **pstate, #type)
```

The `pstate` is the pointer to the private data. The necessary amount of memory (based on data `type`) is allocated during this call. Within this same memory area all the data from the asynchronous request that need to be preserved for some time should be kept.

[7] Talloc's capacity to take over data within the hierarchical structure from a parent and assign them to another. This technique is described in the thesis concerning talloc [3] as well as in talloc documentation [14].

Dealing with a lack of memory

The verification of the returned pointer against `NULL` is necessary in order to identify a potential lack of memory. There is a special function which helps with this check.

```
bool tevent_req_nomem (const void *p, struct tevent_req *req)
```

It handles verification both of the talloc memory allocation and of the associated tevent request, and is therefore a very useful function for avoiding unexpected situations. It can easily be used when checking the availability of further memory resources that are required for a tevent request. Imagine an example where additional memory needs arise although no memory resources are currently available.

```
bar = talloc(mem_ctx, struct foo);
if(tevent_req_nomem (bar, req)) {
   /* handling a problem */
}
```

Listing 3.6: Checking process of memory allocation.

This code ensures that the variable `bar`, which contains `NULL` as a result of the unsuccessful satisfaction of its memory requirements, is noticed, and also that the tevent request `req` declares it exceeds memory capacity, which implies the impossibility of finishing the request as originally programmed.

3.4.2 Binding a Callback to an Asynchronous Request

Callback is a function triggered after the event it is bound to has occurred. It is necessary to prepare for dealing with this request in any situation (with or without any errors) and to pay good attention to setting the request either as a success or as a failure, otherwise the callback will never be triggered at all.

The connection of a request to a specific callback function, usually while also adding some private data by pointer, is simple.

```
void tevent_req_set_callback (struct tevent_req *req,
                              tevent_req_fn fn, void *data)
```

The pointer `data` in this example is usually a pointer to *private data* - a structure containing some data, but it can also be a pointer to another `tevent_req` structure, through which it is possible to access information allocated at the moment of the request's creation.

It is very important to create both callback and also functions that are able to deal with various scenarios which may happen with a request.

- no error - take the action that was originally planned
- request timeout - the handler that was set to deal with the event at first will not be triggered after all, but the callback will be. Tevent may in fact take the callback triggering action twice - once, at the moment when the request was set as completed or failed, and then again when a timeout interval is reached. It is good to be aware of and prepared for this.
- error - handle the error and if necessary, hand the indication of error further.

3.4.3 Accessing Private Data

A tevent request is (usually) created together with a structure for storing the data necessary for an asynchronous computation. For these *private data*, tevent library uses void (generic) pointers, therefore any data type can be very simply pointed at. However, this attitude requires clear and guaranteed knowledge of the data type that will be handled, in advance. Private data can be of 2 types: connected with a request itself or given as an individual argument to a callback. It is necessary to differentiate these types, because there is a slightly different method of data access for each.

There are two possibilities how to access data that is given as an argument directly to a callback. The difference lies in the pointer that is returned. In one case it is the data type specified in the function's argument, in another `void*` is returned.

```
void* tevent_req_callback_data (struct tevent_req *req, #type)
void* tevent_req_callback_data_void (struct tevent_req *req)
```

To obtain data that are strictly bound to a request, this function is the only direct procedure.

```
void *tevent_req_data (struct tevent_req *req, #type)
```

As you can see in the next example, the difference between functions returning pointers to private data of the callback function, lies in their explicit conversion of the data.

```
struct foo_state {
   int x;
};

struct test {
   int y;
};

static void foo_done(struct tevent_req *req) {
   // a->x contains 9
   struct foo_state *a = tevent_req_data(req, struct foo_state);

   // b->x contains 10
   struct test *b = tevent_req_callback_data(req, struct
                                             test);

   // c->x contains 10
   struct test *c = (struct test *)
                    tevent_req_callback_data_void(req);
}
struct tevent_req *foo_send(TALLOC_CTX *mem_ctx,
                            struct tevent_context *event_ctx,
                            ...) {
   struct foo_state *state;
   struct tevent_req *req;
```

```
    req = tevent_req_create(mem_ctx, &state, struct foo_state);
    state->x = 10
    /* ... */
}

void run(TALLOC_CTX *mem_ctx, struct tevent_context *event_ctx) {
    struct test *tmp = talloc(mem_ctx, struct test);
    tmp->y = 9;
    req = foo_send(mem_ctx, event_ctx, ...)
    /* ... */
    tevent_req_set_callback(req, foo_done, tmp);
}
```

Listing 3.7: Getting private data through generic pointers

3.4.4 Finishing a Request

Marking each request as finished is an essential principle of the tevent library. Without marking the request as completed - either successfully or with an error - the tevent loop could not let the appropriate callback be triggered. It is important to understand that this would be a significant threat, because it is not usually a question of one single function which prints some text on a screen, but rather the request is itself probably just a link in a series of other requests. Stopping one request would stop the others, memory resources would not be freed, file descriptors might remain open, communication via socket could be interrupted, and so on. Therefore it is important to think about finishing requests, either successfully or not, and also to prepare functions for all possible scenarios, so that the the callbacks do not process data that are actually invalid or, even worse, in fact non-existent meaning that a segmentation fault may arise.

Manually

This is the most common type of finishing request. Calling this function sets the request as a `TEVENT_REQ_DONE`. This is the only purpose of this function and it should be used when everything went well. Typically it is used within the `_done` functions (for a reminder of the naming conventions, see Section 3.4).

```
void tevent_req_done (struct tevent_req *req)
```

Alternatively, the request can end up being unsuccessful.

```
bool tevent_req_error (struct tevent_req *req, uint64_t error)
```

The second argument takes the number of an error (declared by the programmer, for example in an enumerated variable). The function `tevent_req_error(...)` sets the status of the request as a `TEVENT_REQ_USER_ERROR` and also stores the code of error within the structure so it can be used, for example for debugging. The function returns true, if marking the request as an error was processed with no problem - value `error` passed to this function is not equal to `1`.

Setting up a timeout for request

A request can be finished virtually, or if the process takes too much time, it can be timed out. This is considered as an error of the request and it leads to calling callback.

In the background, this timeout is set through a time event (described in Section 3.3.2) which eventually triggers an operation marking the request as a `TEVENT_REQ_TIMED_OUT` (can not be considered as successfully finished). In case a time out was already set, this operation will overwrite it with a new time value (so the timeout may be lengthened) and if everything is set properly, it returns true.

```
bool tevent_req_set_endtime (struct tevent_req *req,
                             struct tevent_context *ev,
                             struct timeval endtime)
```

Premature Triggering

Imagine a situation in which some part of a nested subrequest ended up with a failure and it is still required to trigger a callback. Such as example might result from lack of memory leading to the impossibility of allocating enough memory requirements for the event to start processing another subrequest, or from a clear intention to skip other procedures and trigger the callback regardless of other progress. In these cases, the function `tevent_req_post(...)` is very handy and offers this option.

```
struct tevent_req* tevent_req_post (struct tevent_req *req,
                                    struct tevent_context *ev)
```

A request finished in this way does not behave as a time event nor as a file descriptor event but as a immediately scheduled event, and therefore it will be treated according the description laid down in Section 3.3.4.

3.4.5 Subrequests - Nested Requests

To create more complex and interconnected asynchronous operations, it is possible to submerge a request into another and thus create a so-called *subrequest*. Subrequests are not represented by any other special structure but they are created from `tevent_req_create(...)`. This diagram shows the nesting and life time of each request. The table below describes the same in words, and shows the triggering of functions during the application run.

Wrapper represents the trigger of the whole cascade of (sub)requests. It may be e.g. a time or file descriptor event, or another request that was created at a specific time by the function `tevent_wakeup_send(...)` which is a slightly exceptional method of creating tevent requests.

```
struct tevent_req *tevent_wakeup_send (TALLOC_CTX *mem_ctx, struct
                                       tevent_context *ev, struct
                                       timeval wakeup_time)
```

By calling this function, it is possible to create a tevent request which is actually the return value of this function. In summary, it sets the time value of the tevent request's creation. While using this function it is necessary to use another function in the subrequest's callback to check for any problems `tevent_wakeup_recv(tevent_req *req)`)

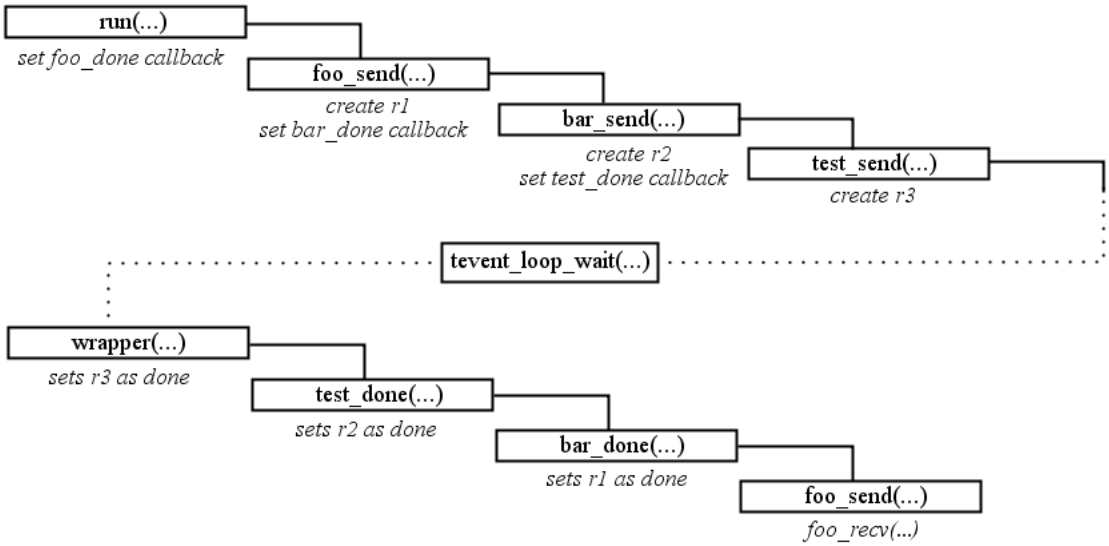


Figure 3.2: Handling of events, [6]

Time	Function	Action
	`run()`	set `foo_done()` callback
	`foo_send()`	*r1* is created; set `bar_done()` callback
⋮	`bar_send()`	*r2* is created; set `test_done()` callback
	`test_send()`	*r3* is created
x	`tevent_loop_wait()`	-
$x+m$	`wrapper()`	*r3* is set as done
$x+n$	`test_done()`	*r2* is set as done; `test_recv()`
$x+o$	`bar_done()`	*r1* is set as done; `bar_recv()`
$x+p$	`foo_done()`	`foo_recv()`

Table 3.2: Sequence of triggering of functions during application run

The functions with the suffix `_recv` are optional and so as not to make the table unnecessarily big, these have been placed into the column of actions.

Aforementioned information about subrequests might seem to be a bit complicated with no real example within this thesis. Comprehensive example containing features from the Section 3.4 is especially by reason of nested subrequests very long and therefore it is located as an example on the attached CD. Description and explanation is placed within the source code itself (`subrequest.c`) and enclosed file `README`.

3.5 Queues of Events with tevent

There is a possibility that the dispatcher and its handlers may not be able to handle all the incoming events as quickly as they arrive. One way to deal with this situation is to buffer the received events by introducing an event queue into the events stream, between the events generator and the dispatcher. Events are added to the queue as they arrive, and the dispatcher pops them off the beginning of the queue as fast as possible [6].

In tevent library it is similar, but the queue is not automatically set for any event. The queue has to be created on purpose, and events which should follow the order of the FIFO queue have to be explicitly pinpointed. Creating such a queue is crucial in situations when sequential processing is absolutely essential for the succesful completion of a task, e.g. for a large quantity of data that are about to be written from a buffer into a socket.

The tevent library has its own queue structure that is ready to use after it has been initialized and started up once.

3.5.1 Creation of Queues

The first and most important step is the creation of the tevent queue (represented by `struct tevent_queue`), which will then be in *running mode*.

```
struct tevent_queue* tevent_queue_create (TALLOC_CTX *mem_ctx,
                                          const char *name)
```

When the program returns from this function, the allocated memory, set destructor and labeled queue as running has been done and the structure is ready to be filled with entries.

Stopping and starting queues on the run

If you need to stop a queue from processing its entries, and then turn it on again, a couple of functions which serve this purpose are:

- `bool tevent_queue_stop (struct tevent_queue *q)`
- `bool tevent_queue_start (struct tevent_queue *q)`

These functions actually only provide for the simple setting of a variable, which indicates that the queue has been stopped/started. Returned value indicates result.

3.5.2 Adding Requests to a Queue

Tevent in fact offers 3 possible ways of inserting a request into a queue. There are no vast differences between them, but still there might be situations where one of them is more suitable and desired than another.

With No Further Possibility of Management

```
bool tevent_queue_add(struct tevent_queue *queue, struct
                      tevent_context *ev, struct tevent_req *req,
                      tevent_queue_trigger_fn_t trigger, void
                      *private_data)
```

This call is the simplest of all three. It offers only *boolean* verification of whether the operation of adding the request into a queue was successful or not. No additional deletion of an item from the queue is possible, i.e. it is only possible to deallocate the whole tevent request, which would cause triggering of destructor handling and also dropping the request from the queue.

Extended Options

Both of the following functions have a feature in common - they return `tevent_queue_entry` structure representing the item in a queue. There is no further possible handling with this structure except the use of the structure's pointer for its deallocation (which leads also its removal from the queue). The difference lies in the possibility that with the following functions it is possible to remove the tevent request from a queue without its deallocation. The previous function can only deallocate the tevent request as it was from memory, and thereby logically cause its removal from the queue as well.

There is no other utilization of this structure via API at this stage of tevent library. The possibility of easier debugging while developing with tevent could be considered to be an advantage of this returned pointer.

```
struct tevent_queue_entry*
tevent_queue_add_entry(struct tevent_queue *queue, struct
                       tevent_context *ev, struct tevent_req *req,
                       tevent_queue_trigger_fn_t trigger, void
                       *private_data)
```

The feature that allows for the optimized addition of entries to a queue is that a check for an empty queue with no items is first of all carried out. If it is found that the queue is empty, then the request for inserting the entry into a queue will be omitted and directly triggered.

```
struct tevent_queue_entry*
tevent_queue_add_optimize_empty(struct tevent_queue *queue, struct
                                tevent_context *ev, struct
                                tevent_req *req,
                                tevent_queue_trigger_fn_t trigger,
                                void *private_data)
```

When calling any of the functions serving for inserting an item into a queue, it is possible to leave out the fourth argument (`trigger`) and instead of a function pass a `NULL` pointer. This usage sets so-called *blocking* entries. These entries, since they do not have any trigger operation to be activated, just sit in their position until they are labeled as a *done* by another function. Their purpose is to block other items in the queue from being triggered.

Chapter 4

Tevent Extension for GDB

To improve development with tevent, an extension for `gdb` was created as a part of this thesis. A script extension written in Python was designed to fulfill basic debugging needs when programming with tevent. Features that are implemented in the plug-in were discussed with Red Hat developers and based on their requests [1]. The reason for choosing the extension for `gdb` is its widespread familiarity among developers and the fact that it is one of the most common debugging tools. The plugin was also tested and works with graphical front-end `ddd`.

The plugin's main features are described here together with the outputs of the debugger, but for deeper knowledge it is recommended to look at the source code of the plug-in directly. The source code of the extension is available on the CD attached to this thesis, or on the internet.

This plug-in was developed and tested on GDB 7.4 and Python 2.7.3.

Extension usage

At first the plug-in must to be loaded into `gdb`. This can be done in two different ways: temporary or constant. The advantage of constant loading is that this extension will then be automatically loaded by GDB whenever the debugger is launched. To do so, you must first create a file `.gdbinit` into your home directory - `~/.gdbinit` (if it does not already exist). Then add a line `source /<path-to-file>/tevent-gdb-extension.py` into the file to tell the debugger to load this script at startup.

The command required to load the Python extension just once should be entered directly within the debugger:

```
(gdb) source tevent-gdb-extension.py
```

Listing 4.1: Including GDB plug-in

Compatibility issues

This extension was tested on different operating systems (Fedora 13, CentOS 5.8, Ubuntu 12.04 LTS and others), distinguish Python (2.x.x) and GDB versions and configurations. This testing revealed different compatibility issues which forced to create two versions of tevent extension. Both of them are placed in appropriate folder on the medium. Lighter version brings the compatibility of more platforms but it is a bit limited in its capabilities. Two functions (`tevent-set-breakpoint-callback` and `tevent-set-breakpoint`) were excluded.

Features of the Extension

After loading the plug-in, the only thing required for its usage is to type specific command. A list of the commands and features which are currently supported in the plug-in follows:

- `tevent-awaiting-requests-num` *`ev`* - shows the number of events waiting to be done in the future. The output is separated into 4 categories (file descriptors, signals, timer and immediate events).
- `tevent-req-status` *`req`* - prints the status of the given tevent request.
- `tevent-set-breakpoint-callback` - sets a breakpoint for every attempt to set up a callback for a tevent request.
- `tevent-set-breakpoint` *`function`* - sets breakpoint(s) for a given function. The symbol '*' (asterisk) may be used as a substitution for any string.
- `tevent-req-callback` *`req`* - shows what callback (if any) is set to a given tevent request.
- `tevent-talloc-parent` *`item`* - shows the parent in talloc hierarchial memory for a given variable.
- `tevent-queue-info` *`q`* - prints information about a queue and its inserted entries.

4.1 Examples of Usage

To present the capabilities of the extension more thoroughly, a quick overview of all the opportunities for working with the extension, with exemplary inputs and output from the `gdb` extension follows.

To acquiant yourself with this extension it is recommended to try its usage with the examples attached on the CD at first to see how tevent works.

Number of Awaiting Events

A *tevent_context* is accepted as the only valid argument. This feature will show up the number of events that are ready to be handled, sorted into groups. As well as the basic number of events, this feature also displays the tevent structure relative to each event.

```
(gdb) tevent-awaiting-events event_ctx
timer_events: 1 (0x804b20)
immediate_events: 0
signal_events: 3 (0x804b5a0, 0x804b5e4, 0x804b610)
fd_events: 2 (0x804b340, 0x804b2d8)
```

Status of a Request

This function discovers whether a request completed successfully or ended up with an error status. It allows as its argument not only the name of a request but also an address in hexadecimal form that points to the allocated memory for the request.

```
(gdb) tevent-req-status 0x804c3c8
Status: TEVENT_REQ_IN_PROGRESS
```

If the address or name is invalid, the extension notifies of this appropriately in the output.

```
(gdb) tevent-req-status 0x804c3aa
Error occurred! Are you sure that '0x804c3aa' is the right
   address of the tevent_req structure?
```

Watching for Callback Setup

This call will inform you and break the application every time an attempt to set a callback (via function `tevent_req_set_callback()`) to a request is about to happen.

```
(gdb) tevent-set-breakpoint-callback
Breakpoint 1 at 0x8048a30
```

Setting up Breakpoint for Specific Function(s)

This allows you to simply set up callback for a whole series of the functions, e.g. `forward_send`, `forward_done` and `forward_recv` by using an asterisk which masks, as usual, any sequence of characters.

```
(gdb) tevent-set-breakpoint forward_*
Breakpoint 1 at 0x8048a30: file client.c, line 330.
static void forward_send (TALLOC_CTX *, struct tevent_context *,
char *);
Breakpoint 2 at 0x254af40: file client.c, line 382.
static void forward_done (struct tevent_req *);
Breakpoint 2 at 0x8049010: file client.c, line 450.
static void forward_done (struct tevent_req *, int *);
```

Showing Request's Callback

Usage of following extension displays whether a callback has been set for a specific request. If it has, the name of the function that is called as a callback is printed to standard output.

```
(gdb) tevent-req-callback preq
No callback was set to the request.

(gdb) tevent-req-callback subreq
Callback function for the request is: 'connect_done'
```

Showing the Parent Node in talloc

This finds a parent node within the talloc memory tree. If such a node has a name it prints it. Naming nodes is optional in talloc and therefore a situation could occur when the parent node does not have a name. In this case, the data type and address where is stored will be displayed.

```
(gdb) tevent-talloc-parent event_ctx
Talloc parent is: 'root' at address 0x8049970
```

Showing Information About a Queue

After creation of a tevent queue it is possible to display information about it - the name of the queue, its status (whether it is running or stopped) and its length, together with addresses for each of the elements stored in the queue.

```
(gdb) tevent-queue-info q
Name of the queue: 'test'
Status: IS RUNNING
Length: 2
Node(s): 0x804c880, 0x804c8d8
```

Chapter 5

Conclusion

This thesis has acquainted readers with the concept of the event-based programming paradigm, its advantages and disadvantages, and has introduced several libraries that support programming with events, as well as pointing out their capabilities.

A presentation and investigation of the tevent library with a description of its capabilities followed. A lot of attention was paid both to the single event approach, and to the principle of creating nested subrequests. The question of subrequests is a major interesting feature of tevent library and is among others widely used in projects Samba and SSSD. These features of the library were not previously satisfactorily documented and I am pleased to make tevent more accessible for others who might start working with it. Improvement of official documentation on project homepage is being discussed with developers.

At the end of thesis, an extension for GDB debugger was introduced. This plugin was created to facilitate work with tevent library by fulfilling developers' suggestions gathered from the open mailing-list of the SSSD project.

Finally, benchmark testing of libraries libevent and libev with tevent was presented. This comparison has demonstrated the performance of tevent library in contrast to the more widely used and long-term-developed libraries.

Bibliography

[1] [SSSD] [tevent] GDB extension for tevent library [online]. https://lists.fedorahosted.org/pipermail/sssd-devel/2013-March/013809.html.

[2] What's the difference between libev and libevent? [online]. http://stackoverflow.com/questions/9433864/whats-the-difference-between-libev-and-libevent.

[3] Březina, Pavel. *Talloc - a hierarchical memory allocator*, spring 2012.

[4] Frank Dabek, Nickolai Zeldovich, Franks Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 2002 SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[5] Dickson K. W. Chiu and Wesley C. W. Chan and Gary K. W. Lam and S. C. Cheung and Franklin T. Luk. An event driven approach to customer relationship management in an e-brokerage environment. In *36th Hawaii International Conference on System Sciences (HICSS36)*, page 10, 2003.

[6] Ferg, Stephen. *Event-Driven Programming: Introduction, Tutorial, History*[online]. http://heanet.dl.sourceforge.net/project/eventdrivenpgm/event_driven_programming.pdf, 2008/02/08. (visited on 10/11/2012).

[7] Han, Sangjin. Scalable event multiplexing: epoll vs. kqueue [online]. http://www.eecs.berkeley.edu/~sangjin/2012/12/21/epoll-vs-kqueue.html.

[8] Hansen, Stuart and Timothy V. Fossum. *Event Driven Programming* [online]. http://www.cs.uwp.edu/staff/hansen/EventsWWW/Text/Events.ps, 2011.

[9] Lehmann, Marc. Benchmarking libevent against libev [online]. http://libev.schmorp.de/bench.html.

[10] Lehmann, Marc. Libev [online]. http://software.schmorp.de/pkg/libev.html.

[11] Lemon, Jonathan. *Kqueue: A generic and scalable event notification facility* [online]. http://people.freebsd.org/jlemon/papers/kqueue_freenix.pdf, 2000.

[12] Maharjan, Pasa. *Comparing and Measuring Network Event Dispatch Mechanisms in Virtual Hosts* [online]. http://dspace.cc.tut.fi/dpub/bitstream/handle/123456789/21143/maharjan.pdf, 2011.

[13] Mathewson Nick and Provos, Niels. Libevent – an event notification library [online]. http://libevent.org/.

[14] Samba. Talloc documentation [online]. http://talloc.samba.org.

[15] Samba. Tevent documentation [online]. http://tevent.samba.org/index.html.

[16] Tucker, A.B. and Noonan, R. *Programming languages: principles and paradigms.* McGraw-Hill, 2002.

[17] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazières, and Frans Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003.

Appendix A

DVD contents

- `benchmark/` - performance results
- `examples/` - source codes describing usage of tevent.
- `extension/` - GDB extension
- `thesis/` - contains this document and LaTeX source codes of this bachelor thesis

Appendix B

Comparison with libevent

A brief introduction to libraries that work with events was presented in Section 2.1 where a summary table comparing all the libraries in terms of their OS support and polling mechanisms was provided. The information presented in this appendix will focus on a more detailed comparison of libevent and tevent, for which not only their supported platforms and features will be examined. In the graphs and comparisons the libev library is also included, to show the performance differences that result from libev optimization. For a more complex overview the brief introduction should also be read.

Just to summarize, all three libraries (libevent, libev and tevent) support high performance *epoll* and are capable of running on Unix type systems, Mac OS X, *BSD, Windows. Tevent does not yet officially support the kqueue mechanism, but this is about to changed. Once tevent introduces kqueue support, all three libraries will also be equivalent concerning the operating system Solaris. The main difference that makes libevent more efficient on Windows platform is its ability to work with IOCP (Input/output completion port), which offers more throughput for asynchronous I/O than just `select()`.

Specific features of libev which are not commonly implemented by other libraries and are missing in tevent's range include: watchers that monitor either child process or process labeled with PID in general, time events not only for relatively expressed time or timeouts but also for absolutely specified time, which may behave like *cron*; monitoring of filesystem objects (with the ability, above standard readability or writability, also to change the attributes of files); and the option to interconnect several event loops together and communicate between them.

In comparison to other libraries, libevent offers a framework that enables the user to control tasks regarding network communication much more easily. Further features that accompany this possibility are SSL, rate limits, and support for protocols such as HTTPS, DNS.

It is noticeable that libevent tries to provide a variety of capabilities and a complex solution. On the other hand, libev only implements the event library itself, with the aim of creating the highest possible level of performance [2].

There is a clear difference between the features included in tevent and those both of libev and of libevent. Tevent released its first version in Autumn 2009, compared with libevent whose first release was issued in 2000. Nine years further development has enabled libevent to provide both more features and better performance (libev, which is modelled on libevent, was introduced in 2007).

B.0.1 Benchmark Setup

This part of the thesis examines and compares the performance of libevent and tevent. No previous benchmark of tevent and another library is known, and therefore a performance test was run within this thesis to find out how efficient tevent actually is.

All tests were run on the following computer:

CPU:	Intel® Core™ i3-3225 @ 3.30 GHz
Memory:	8GB DDR3 667 MHz
OS:	Ubuntu 12.04 LTS 64-bit
tevent:	tevent-0.9.17
libevent:	libevent 2.0.16
libev:	libev 4.11

There is a known comparison between libev and libevent, which is provided and updated by the libev developers on their website. This test was modified and used for this thesis as well.

The benchmark consists of creating pairs of sockets; event watchers are then set, and at the end a smaller number of active clients send and receive data to a subset of those sockets [9]. The time measurements cover the total time spent within a function, which includes setting up handlers for an event, as well as the event loop itself.

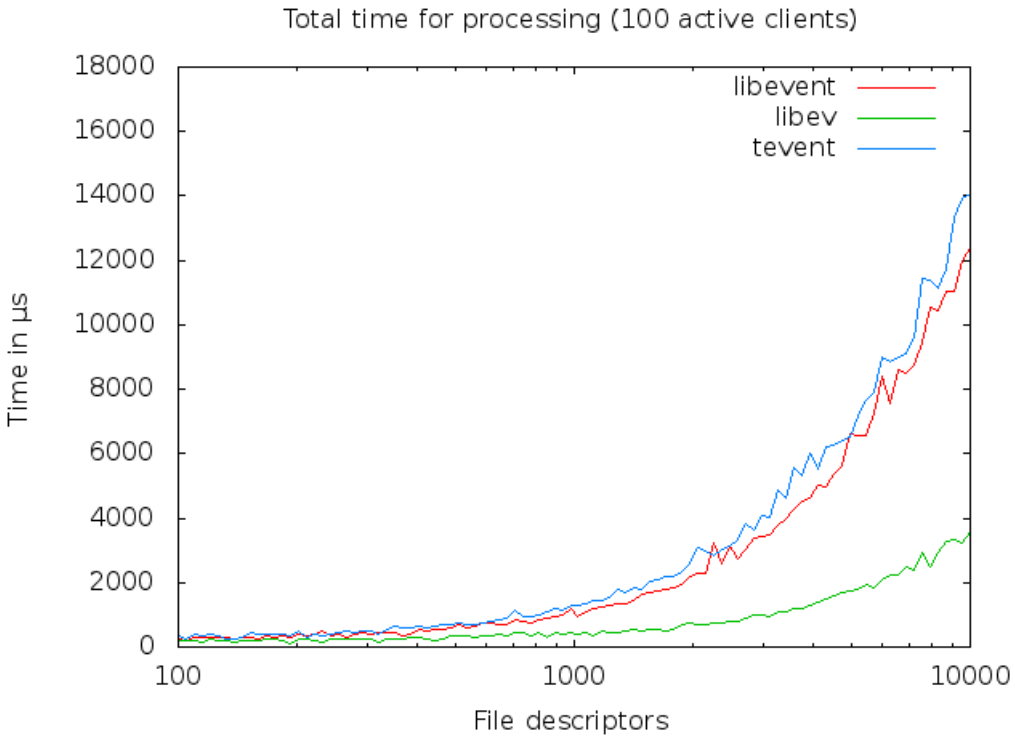


Figure B.1: 100 active clients

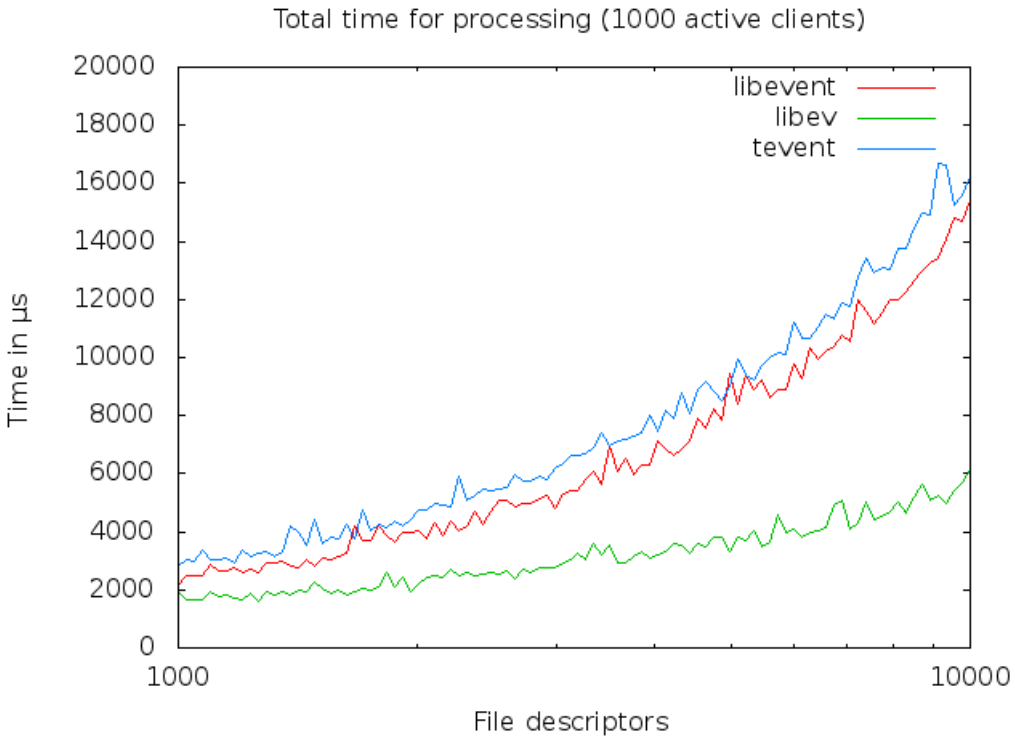


Figure B.2: 1000 active clients

The test was carried out for 100 and 1000 active clients. The number of file descriptors opened was gradually increased up to 10000. An epoll interface was used because this is the most highly performing mechanism supported by both the operating system and the libraries (for more information about system calls, see 2.2).

The source files (included on the attached CD) were compiled with GCC version 4.6 with optimization `-O3`.

```
gcc tevent_bench.c -o tevent_bench -ltevent -ltalloc -levent -O3
gcc libevent_bench.c -o libevent_bench -levent -O3
gcc libev_bench.c -o libev_bench -lev -O3
```

The graphs show both variants of 100 and 1000 active clients, which each perform 1 input and 1 output operation (`send()` and `recv()`). The benchmark was run several times for each adjustment and the value presented in the graph is the arithmetic mean of all values obtained for the specific configuration.

B.0.2 Conclusion

The difference between the libraries rises with the number of file descriptors, and clearly shows that the libev library is more efficient when dealing with large quantities of file descriptors. The breakpoint at which the libraries start to be clearly distinguishable from one another is at around 1000 file descriptors (seen on the graph with 100 active clients).

The performance difference between libev and libevent occurs for several reasons. The libev developers have a unique attitude in trying to achieve the highest performance library focused on events, and they have been successful in this goal. Libev also takes a different

attitude regarding working with epoll calls, compared to tevent or libevent. Further reasons for the differences in performance between libev and libevent are mentioned on the libev benchmark website [9]. The results in the test I have performed here correspond with the data presented on libev's website.

More interesting and, above all, new are the results of the test on tevent library, which has not previously been benchmarked. Tevent library keeps pace and is just a little slower than libevent, although it has different memory management provided by talloc library. Tevent is approximately 13% slower than libevent in terms of total time. If only the time for event processing (event loop) itself were counted, and the time consumed by registering the event handler were to be excluded, the difference between the libraries would be about 20% (this is not presented in graphs).

Looking at these numbers, it is obvious that tevent is not as fast as libevent. However, taking into account that the time spent processing events is in milliseconds, the results in the time for actual operations differ in the third decimal place at most. Use of tevent therefore could be considered, and the performance difference should not be seen as a reason not to choose tevent over libevent.