Master Thesis
Software Engineering and Distributed Systems

(A)synchronous Programming and (Non-)blocking IO

An approach towards a scalable lock-free asynchronous event loop with encapsulated blocking capabilities

Supervisor:
Dr. Alexander Lazovik
a.lazovik@rug.nl

Author:
Edwin-Jan Harmsma
edwin-jan@harmsma.org

Second Reviewer:
Dr. Rein Smedinga
r.smedinga@rug.nl

September 10, 2012

High performance server systems tend to increasingly rely on asynchronous and non-blocking event loop architectures. The primary reason for this is that non-blocking programming structures are often more scalable with respect to the number of concurrent open connections that a single machine is able to handle. This because a single threaded system will usually have a significant lower memory consumption since no programming stack is allocated for each open connection.

However, the use of a pure asynchronous architecture has a very negative impact on the programming complexity. Applications that, for example, use a traditional callback structure automatically result into very complex state machine implementations inside both the application logic and IO interaction logic layers. The reason for this is that the developer is burdened by the responsibility to control all state transitions and maintaining the active state between IO operations since no information can be stored on the programming stack. This also results into an increased software coupling between components, and even causes that not every protocol implementation can be parsed in an asynchronous system.

In this research, two different hybrid frameworks were created that try to combine the scalability benefits of an asynchronous architecture with the ease of development of synchronous programming. In addition, both approaches remove a major part of the concurrency complexity involved in normal multithreaded environments by using non-preemptive scheduling instead. The use of coroutines in combination with future objects allows the developer to write normal synchronous code that is still able to parallelize certain parts of the communication.

The first approach uses the Scala Continuations library that provides language support for suspending method calls. This resulted in a proof of concept that allows the developer to write synchronous code while still being able to control the asynchronous behavior. However, this approach is still incompatible with blocking stream libraries. Instead, the second approach solves this incompatibility issue by using low level context switching which allows to totally encapsulate the continuation-behavior of the system. This resulted into a framework that allows developers to use all existing blocking API's for IO interaction while the stack allocation overhead is still minimized.

Contents

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this master thesis. I want to thank my supervisor dr. Alexander Lazovik for giving me much freedom with respect to the direction of the research. Especially the brainstorm sessions during our meetings were always a source for new inspiration. In addition, the inspiring lectures about the design of the `IOStream` library during the C/C++ course given by dr. Frank Brokken ensured that I was able to implement the new `streambuf` implementation without any headache. Finally, I want to thank dr. Rein Smedinga for doing the second review.

1. Introduction

'The debate between threads and events is a very old one' states a popular research article published in 2003 [66]. Now, in 2012, it seems that event based systems gain more and more popularity in, for example, the web application field. For instance, the popular *Node.js* project tries to enable the event based Javascript structure inside server systems, arguing that it is designed for 'easily building fast, scalable network applications' [22].

'Ease of use', 'fast' and 'scalable' are not unique claims in the modern application server market. Also, there is a typical cohesion between *developer friendliness* and *scalability*, and often *performance* or *speed* is also mentioned in the project slogans. Many projects claim to be both scalable and developer friendly [22, 20, 42], but the natural tension between these two key drivers is never mentioned explicitly in any project we are aware of. This tension is mainly caused by the difference in interest of both drivers. Scalable means in this case that as few resources as possible should be used, while the ultimate goal of developer friendliness is to have the feeling of unlimited memory and CPU resources.

Traditional *thread-per-connection* architectures are not highly scalable because they suffer from trashing due to many large (but for a major part unused) stack frames that must be kept inside the application memory [66, 4]. In contrast, in an event based system, the user - often supported by a garbage collector - is responsible for maintaining the programming state inside the memory. This introduces additional programming complexity that especially becomes difficult when combining multiple events.

1.1. Events happen when they happen

... and that is also their biggest problem. A problem that can probably be best described by considering a part of a mobile application that was required to (1) enable the filesystem, (2) enable the GPS receiver and (3) send an HTTP request over the Internet. Parallel execution of these three operations was required to maintain a reasonable startup time of the application. Because the application was implemented in the PhoneGap framework, the used programming language was Javascript, and thus an event based implementation was forced by this platform.

Because all three operations must be performed in parallel, the responses are actually only relevant at the moment all these events have been fired. The problem is that all requests expect one or more callbacks that are called by the underlaying Javascript engine in any possible order. For example, the `enableGPS(onSuccess, onError)` function expects two callbacks of which only one is fired. The application is only able to succeed

if all three operations have fired the `onSuccess` callback, and in case of one or more `onError` events it should just display an error message.

The most obvious solution consists of three global variables that represent the state of each operation, and whenever all variables contain a valid state a callback function is called to indicate that the initialization phase is finished. An actual implementation of this task resulted in 65 lines of code where the global control of state was spread among five different callback functions that control the three state variables. This results in very unmaintainable code, for instance, imagine the affected code range if only one parallel operation would be added.

This is just a simple example of how event based (and in this case callback based) programming can highly complicate code structures. An extensive programming analysis is provided in chapter 3.

1.2. (A)synchronous and (non-)blocking

The terms asynchronous and synchronous programming are often used in relation to the terms non-blocking and blocking IO, respectively. This because these principles are often used in conjunction with each other, while this is not strictly required. (A)synchronous programming is the high level implementation of distributed communication, while (non-)blocking IO refers to the underlaying implementation of the low level network interaction.

The difference between the two contrary principles is also similar. The read principle of non-blocking and blocking IO can be stated as 'There are N new bytes available at connection X' versus 'Read N bytes from the input stream of connection X', respectively. This is analogous to the difference between an asynchronous and synchronous request: 'Send something, do something else, and receive something (in varying order)' versus 'Send a request and wait for the response'.

As mentioned in the introduction of this chapter, from the performance perspective both asynchronous and non-blocking programming result in the same implementation issues as well. Synchronous programming and blocking IO are able to use the programming stack, while the developer itself is responsible for maintaining the program state with asynchronous programming and non-blocking IO. Although, one of the two existing architectural approaches given in section 2.2 partially hides the issue of maintaining the programming state.

The memory usage of threads in case of blocking IO is actually always the limiting factor of server systems [71]. This problem increases when connections stay open for a long time. Though, some benchmark attempts on the Internet claim that blocking IO is even more scalable and faster than non-blocking IO. But in this case they also often assume short-lived connections, and manually decrease the stack size of the threads [71]. These arguments for blocking IO are completely valid, but this project especially aims at the problem of long open connections. For example, for real time updates to connected clients. The upcoming section will elaborate on this discussion by giving the benefits and liabilities of different server architectures.

Finally, it should be mentioned that a high traffic server is not the only system that could benefit from asynchronous (and event based) communication. For instance, Google's Chromium project also uses an asynchronous message bus that is responsible for all browser connections and communication between the different tabs [5].

1.3. Server architectures

The C10K problem describes different IO strategies for servers to go ahead the 'magical' 10.000 concurrent connections barrier [4]. This section is intended to bring these abstract descriptions to a more understandable way of reasoning. This is done by describing seven different server architectures. The following paragraphs provide the most important benefits and liabilities, respectively indicated by + and -. Neutral aspects are indicated by +/-. The last paragraph will conclude this overview, and gives the argument why the non-preemptive scheduling approach gets the main research focus in this project.

Process per connection The server socket forks every incoming connection so it can run in a separate process. In this way, the listener socket is able to handle new connections while all connections can do their own tasks independently from each other.

- Context switching is heavy: Processes do not share any application memory (except the instruction memory in some cases), thus all memory must be swapped for every switch.
- Communication between connections always requires inter-process communication because there is no shared data between requests.
- Operating systems are not able to handle a high amount of processes, e.g. Windows XP is able to handle 8400 [34]. However, before this becomes a problem, context switching already drops down the performance completely.

+ Utilizes multiple cores.
+ Preemptive scheduling, all connections have a fair chance to complete their tasks.

Thread per connection A single process and a thread per connected client. The main process creates a thread for every incoming connection.

+ Context switching requires less overhead than for processes, only the stack and program counter must be replaced for every swap. The shared memory is shared for all threads in this process.

- Operating systems are not able to handle a high amount of threads inside a process, 55000 for a 64-bit windows machine [34].
- Requires locking of writes to the shared data segment. Synchronization is a very hard problem because it might easily lead to dead locks or race conditions.

+/- Thread creation time could be solved by using a fixed thread pool.

+ Utilizes multiple cores.

+ Preemptive scheduling.

Asynchronous Callbacks Because most server side requests are heavy IO-event based, it is efficient to let the operating system decide when the server process should handle the IO. This is possible by using an IO-event notification mechanism that is provided by several operating systems (e.g. by select, poll, kqueue, epoll systems calls). As soon as the operating system detects some incoming data it will notify the server process that is waiting for events inside an event loop.

+ Can easily handle 1k of open connections, depending on the number of incoming events [2].

- Highly complicates programming the normal programming flow, especially if the results of multiple requests must be combined because the callback is not allowed to wait or block. Additionally, buffers should be used for writing to connections.

- Everything that could potentially block (e.g. disk IO) must be handled by the event loop to prevent blocking the entire process.

- Non-preemptive scheduling, one long running request could starve all other pending requests.

- Single threaded, does not benefit from multiple cores.

+ No synchronization and locking problems, everything is single threaded and can just write to the shared memory without explicitly acquiring mutual exclusion. Though, thread-safety should be taken into account in case IO operations are performed within a computation.

IO notification and a thread per request. The above two approaches could be combined. In this case all incoming messages are put inside a queue and (a fixed amount of) threads will pop out messages to handle the requests.

+ Can handle a high amount of open connections.

+ Utilizes multiple cores.

- Same synchronization issues as with a thread per connection. However, the solution below could solve this problem in some cases.

+ Preemptive scheduling between running requests, however, if the a fixed thread pool is used starvation might again be a problem.

- Same context switching overhead, however, there are usually less threads and thus less wasting of resources.

Thread per Actor The problems above can be solved by the Actor pattern [63]. In this case, there is a message queue between the IO bus and the thread that is allowed to perform some computations. Mutual exclusion inside the Actor is not required because only one thread is allowed to execute Actor computations at a time and Actors are not allowed to maintain a shared state with the outside world. It should be mentioned that the message queues between actors and the IO bus still have to ensure this mutual exclusion.

- Non-preemptive scheduling between computations of one Actor. This way an actor should never spend too much time on a single computation.

+ No locking and synchronization issues within an Actor.

- An actor is not allowed to maintain any shared state with other parts of the system.

+/- Multiple cores are utilized as long as there are multiple Actor instances. Long computations within one Actor do not benefit from the use multiple cores.

+ Context switching overhead is minimal as long as the number of threads stays low.

Closures instead of normal asynchronous callbacks Some languages supports closures, which is a piece of code, usually a simple function, that can be executed with a snapshot from (or reference to) the variables in the creation context. The most famous project that is currently taking the benefits of this approach is node.js [22]. This has exactly the same benefits asynchronous callbacks, but makes programming a bit simpler for the developer. This principle could for example be combined with *future objects* to allow easy chaining or combining of multiple events.

IO notification with non-preemptive scheduling All architectures with normal threads except the thread per Actor result in hardly solvable synchronization problems. This could be solved by using preemptive scheduling instead.

+/- Optimal performance, CPU and IO is utilized in the best way. However, a single instance uses only one CPU, which could be solved by using multiple shards of one application.

+ No synchronization or locking issues. However, developers should still treat writing to the global/shared memory with care between IO operations, i.e. only commutative operations can be written immediately to a global/class member variable. Keep in mind, that this is small problem and can often easily be solved by storing the data temporary in a local variable on the stack. And if locking is really required, it can just be performed by a normal variable instead of a real lock, which is probably cheaper because disabling hardware interrupts is not required.

- Same as for the callback and closure architectures, application code should not contain any normal blocking code, and long running computations might let other task starves as well.

+ Low context switching overhead: Only switch context if a request really needs to block (i.e. waiting for another request). Additionally, small (or growing) stacks can be used because usually only a small amount of data will be stored. Compiler optimizations could highly improve this performance [66].

+ Easy programming, just the normal linear programming flow, including the use of normal exceptions.

+ Asynchronous sub-requests are possible, which allow to parallelize requests inside a single programming context without starting a new context or thread.

Conclusion The *IO notification with non-preemptive scheduling* architecture hides many of the painful developer issues that occur in a preemptive threaded architecture. In addition, it also removes a major part of the context switching overhead involved with normal threads. The expectation is that this could improve the system performance and scalability. Though, memory consumption might still influence the overall performance in a negative way compared to the closure and callback approaches.

The IO notification with non-preemptive scheduling attempt is not yet 'proven' and 'widely used'. Although, the mature GNU Portable Threads project [16] already shows that non-preemptive scheduling itself can improve the system performance. However, in contrast to the requirements of this project, the GNU project does only aim at the scheduling functionality, instead of also focussing on the scalability aspect of many open connections. In addition, an important driver of this new project is developer friendliness. This for example means that developers should be able to easily parallelize outgoing requests and still having maintainable synchronous code.

A more recent approach to increase the developer friendliness of the event loop architecture is provided by the *gevent* project [21] that relies heavily on the freedom a scripting language offers. Our project assumes that it might be very useful to have the type safety of a static typed language and thus being able to catch many errors compile-time. Also, the newly created project has the important intention to totally hide the creation of (non-) preemptive threads for the developer, which is not the case for both the pth thread and gevent projects.

Finally, programming language support for coroutines as Scala provides [61] does re-implement the old non-preemptive scheduling paradigm in a new and fresh way. This actually shows that there is a demand for a scalable system that still has the easy way of reasoning blocking systems have.

1.4. Key concepts

Every (research) project has an underlying reasoning behind the actual goals that do not necessary rely on (already) proven facts. This section intends to highlight these implicit arguments and their rationals that will play an important role through the entire project. The upcoming section will state the actual goals of the project.

- Overall scalability in a project is achieved by having **multiple running processes of the same server application**. Multiple processes can run on the same machine as well to utilize multicore systems. Requests to the server application can be partitioned and replicated, this should balance the load in an equal way between server instances. Please note that the actual partitioning or replication is far beyond the scope of this project. *Rational:* Every scalable project has to grow among multiple physical machines, and should thus be partition tolerant among separate running processes. If this scalability goal is achieved, then this same principle can be used to benefit from multiple cores inside one machine. This gives more fault-tolerant and scalable solutions (i.e. one dead process is only responsible for a small part in the entire system).

- Connections stay open longer and longer because application logic increasingly tends to rely on real time updates. The fact that it is now not necessary to start a new connection for every request or push-update is another benefit.

- There is a single event-loop in the entire system that is controlled by **one main thread**. Additional preemptive threads are not allowed, *rational*:
  - Virtual servers (e.g. cloud instances or normal virtualization) usually have less resources and thus cannot handle thousands of threads [4].
  - From the scalability point of view it is better to have X lightweight processes instead of one single heavyweight process that must run on a single machine. In addition, in the first case it is possible to disable servers during low demand periods, while in the last case an entire server should be moved to a smaller instance. Furthermore, having 32 micro instances instead of 1 extra large instance at for example Amazon EC2, provides much higher CPU peak capacity [1] and is able to handle fault tolerance in a better way. Though, in this last case there is always more memory consumption overhead (i.e. 32 small operating systems must be in memory as well).
  - Locking would be required whenever multiple preemptive threads would communicate with the same underlying event loop. This would waste processing time and also complicate the entire implementation if it is not clear whether depended requests would be handled by different threads. In this case, complex concurrency issues would highly complicate both the framework and application code of the developer.
  - External libraries are often not thread-safe, and assume that only one thread is accessing the library at the same time.
  - Even standard libraries are usually not thread-safe (e.g. STL *vector* in C++ or *java.util.Vector* in Java).

[1] Micro instance ($0.020 per Hour for 613MB and *up to* 2 compute units) and Extra Large instance ($0.640 per Hour for 15GB and 8 compute units). From `http://aws.amazon.com/ec2/pricing/`.

- Conceptually, threads will ensure that resources are used in a more optimal way, i.e. continue processing if another thread is pending on IO operations. The goal of this project is to use a scalable event loop architecture, so normal blocking of threads is not possible at all. Under this assumption the previous stated argument is now irrelevant.

  A second argument for using preemptive scheduled threads is to have a more equal response time for all requests. This argument becomes irrelevant as well because the system is designed for projects that are really IO bound. Because an IO bound system only handles short computations, it will not delay subsequent requests very much.

- In cases where long running computations are required by the application domain, preemptive scheduling is still unavoidable in order to maintain a good response time for each individual request. Also, an external library or some other legacy code could use threads internally. In both cases it is important that the event loop library still provides **thread-safe access** to these systems.

1.5. Project goals

The primary goal of this project is to make implementing distributed systems easier for the developers and framework designers while still having the scalability benefits of an event loop architecture. For this reason the most important key drivers are **Developer Friendliness** and **Scalability**, where the developer friendliness has the highest priority. Though, the ultimate goal is that the scalability of the number of concurrent open connections will not be impacted significantly.

Additionally, the project aims on lightweight systems (e.g. virtual and cloud instances) and the designed framework will be single threaded because of the reasons explained in the previous section.

Main research question *How can the complexity of asynchronous programming be improved without affecting the scalability in a significant way?*

Secondary research question *Is it possible to combine the benefits of an event based architecture with the ease of development that traditional stack-based programming provides?* This question emerged from the observed issues in chapter 3. In this chapter it is shown that pure blocking code structures provide in some cases a far more readable and maintainable implementation compared to the other approaches.

2. Event loop architectures

The intention of this chapter is to discuss the event loop architecture and the current state of the art of some existing event loop projects.

2.1. Event loop structure

The following three simple lines of code probably express the entire concept of an event loop in the most easy way:

```
while (continue)
  event := getNext()
  handleEvent(event)
```

An important remark that should be made before this structure is discussed in more detail, is that this mechanism is also commonly used in cases without any relation to asynchronous IO handling. For example, a lot of GUI toolkits use the same structure for handling incoming widget events from other GUI threads, and even network engines might use this pattern in several different places. A network engine could for instance have an event loop for polling the IO events, and in addition provide an event loop to the outside world that allows to pump messages from the internal queue. In this case, the event loop just provides the layer between the event handlers and the actual event provider.

In this document, in contrast to all the above implementations, any upcoming references to the term 'event loop' do refer to the main loop inside the IO frameworks. The responsibilities of this main loop often include dispatching the IO events, timer events and other interrupts or other spontaneous events [57].

2.1.1. Interaction with the event loop

An important observation that already can be made is that the event loop pattern always results into an architecture where the event loop is the core component of the entire system. In other words, from the programming perspective, usually all control flows start with the event loop on top. The event handlers of the loop can be seen as the repeatedly executed `main` functions of normal applications. Now, whenever an event arrives, the handler function will be executed with the data of that specific event. And at the moment another similar event arrives, the same handler function is just executed again.

From the architectural view point, the above structure always returns in a *state machine architecture* for the entire application logic. This because this logic has to maintain

the active state manually, since it cannot be assumed that the events will arrive in linear order. And from a low level point of view, the content of the programming stack is destroyed after every handler call, so it is also not possible to maintain the state on this stack. Also, context switching between events of different connections is performed manually which requires that every connection maintains its own state manually (e.g. authentication information or status information). This means in practice that if the implementation will be a little bit more complex than simple forwarding of data, it will immediately result into complex and large state machines.

An example of such a forced state machine implementation, is the asynchronous implementation of the SSL and TLS protocol. These well known protocols are normally implemented as wrappers around normal sockets, and just encapsulate their entire functionality by just providing, for example, a *SSLOuputStream* and *SSLInputStream* to the rest of the system. In this case, the stream classes will maintain the state of the SSL/TLS connection over time. In addition, these classes also allow to hide potential re-handshakes of the underlying system for the actual callers of the `read` and `write` methods. This should not by that different in the asynchronous way, but the reality is different for, for example, OpenSSL [55] [51]. And before the library supported non-blocking IO it was even almost impossible to use this library in combination with event loop controlled communication.

2.2. Reactor and Proactor patterns

The term 'Event loop' is commonly used in conjunction with the 'Reactor' [69] and 'Proactor' [64] design patterns. These patterns offer two different techniques to multiplex IO events, which is the most important part of an event loop that is finally 'used' by the developer. The differences between these two patterns can be as best explained by describing a potential read event respectively for the *Reactor* and *Proactor* pattern[1]:

1. An event handler declares interest in I/O events that indicate readiness for read on a particular socket
2. The event demultiplexor waits for events
3. An event comes in and wakes-up the demultiplexor, and the demultiplexor calls the appropriate handler
4. The event handler performs the actual read operation, handles the data read, declares renewed interest in I/O events, and returns control to the dispatcher

[1]Descriptions taken from [6]

1. A handler initiates an asynchronous read operation (note: the OS must support asynchronous I/O). In this case, the handler does not care about I/O readiness events, but is instead registers interest in receiving completion events.
2. The event demultiplexor waits until the operation is completed
3. While the event demultiplexor waits, the OS executes the read operation in a parallel kernel thread, puts data into a user-defined buffer, and notifies the event demultiplexor that the read is complete
4. The event demultiplexor calls the appropriate handler;
5. The event handler handles the data from user defined buffer, starts a new asynchronous operation, and returns control to the event demultiplexor.

The difference between these two the design patterns is a matter of the responsibility of when an event is fired. The reactor pattern always fires whenever *an* IO event arrives, and the developer is itself responsible for handling the data whatever this data contains. In contrast, the proactor pattern includes this responsibility in the pattern itself. In this case, the developer only has to register a read request of N bytes, and the event loop will only fire as soon as this request is actually *completed*. Thus, from the developer point of view, the proactor pattern allows to write the actual IO request and responses with less (complex) code.

2.2.1. Multithreading

After an event fires, an *event handler* should perform some implemented actions. By default, if the event loop would execute such a handler function it will do it in same thread. However, it is also possible to dispatch such a handler call to a separate thread, or it is even possible that a set of concurrent worker threads query the event loop simultaneously. In this last case, the event loop must of course be implemented in a thread-safe way. The most important benefit of using thread dispatching for handler functions, is that it is now allowed to use blocking function calls inside the function because only the active thread will block. However, this should be used with care as explained in the next section.

Long open connections Lets assume a system with many active connections, where every active connection just proxies all incoming and outgoing requests to an internal publish subscribe system. This system is shown schematically in Figure 2.1, the proxy could for instance also be the business logic of a web application. A potential implementation of the event handler of this proxy is shown in Listing 2.1. Developers who have worked with server systems where connections are not allowed to stay open for

more than, for example, 30 seconds will directly see that the while loop might become a serious problem. However, in case of an old fashioned blocking and thread based server system, this could just be correct code. Now, the question is if this code actually good or bad.

The most important goal of this project is scalability, and still providing a developer friendly environment. This last aspect is in this example surely approved, the handler code is simple and just requires normal 'top-down thinking' for the developer. Though, the scalability aspect is not satisfied, because the example requires a thread per connection.

A dangerous aspect of this is that downfalls in performance will only be detected during real stress testing. The system could specify that its throughput is, for example, 50 million of messages per second, but such hidden 'programming mistakes' will influence this number in a very negative way when the number of connected clients grows to a certain level. Furthermore, a risky aspect is that event loop frameworks often use pre-allocated thread pools (i.e. with a fixed number of threads) to make the dispatching part more efficient. In this case, the problem becomes even worse because the entire system will block at a given moment. This because all worker threads are at that moment blocking on the IO operation.

In summary, to use threads for simple 'top-down code' really improves developer friendliness. And to use threads for spreading communicational load among multiple processor cores is also a good idea. However, developers should never fall back into pure blocking applications. The asynchronous behavior must be preserved in order to stay scalable, and to prevent that developers will fall back into a *thread-per-connection*. Finally, threads are a nice way to still use *some* blocking code and prevent starvation among other handlers, but long subsequently blocking operations in the same thread can harm the performance in a very negative way.

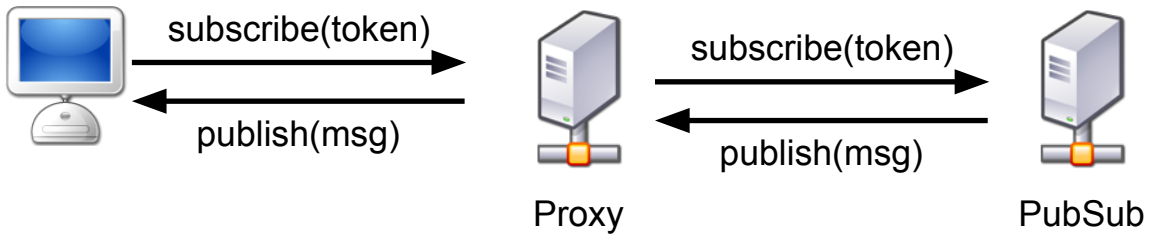


Figure 2.1.: Example of a system where the business logic proxies requests to an internal publish subscribe system.

Listing 2.1: An obvious example of how thread dispatching could easily mislead the developer and completely destroy the scalability a system.

```
1 // Assume the variable pubSub, which maintains an connection with
2 // an external publish subscribe serve.
3 public void onIncoming(RegistrationMessage msg) {
4   pubSub.subscribe(msg.token);
```

```
5    while (!finished)
6    {
7      // Blocking call to remote publisher
8      Publication pub = pubSub.receive();
9      // After received, reply back to connected client
10     client.write(new PublicationReply(pub));
11   }
12 }
```

2.2.2. Event providers

The reactor has to query the operating system for IO events, this is performed by different and often platform depended systems calls which are explained in the next section. In real projects these platform depended calls are often encapsulated by a separate network library that ensures the portability of application. Popular examples of these libraries include libevent [33], libev [32] and JBoss Netty [42].

Selectors

The following table gives an overview of the most commonly used event selectors with their corresponding properties and supported events. The figures displayed after will give a clear indication about the performance measured during the runtime.

Selector	**Properties**	**Supported Events**
Unix `select()` [49]	Monitor multiple file descriptors, indicates when a file descriptor is ready to perform some operations without blocking. Number of monitored file descriptors is fixed, and descriptors and interests must be passed to `select()` every time again. The `select()` call has a complexity of O(n) with respect to number of monitored file descriptors.	Developer is able to pass sets of *read*, *write* and *exception* file descriptors to the system call, to indicate which interest should be monitored for each file descriptor. In additional, a timeout can be specified for the maximum amount of time that the call will return.

Unix `poll()` [44]	Similar to `select()`, but now without a fixed number of monitored file descriptors. `select()` and `poll()` are usually even implemented as wrappers around each other. The `poll()` call has a complexity of O(n) with respect to number of monitored file descriptors.	The following events are at least be supported: **POLLIN** There is data to read **POLLPRI** There is urgent data to read **POLLOUT** Writing now will not block **POLLRDHUP** Stream socket peer closed connection, or shut down writing half of connection **POLLERR** Error condition for output) **POLLHUP** Hang up for output only
Unix `epoll()` [8]	Is a variant of `poll()` that scales well to large numbers of monitored file descriptors. The functionality of `epoll` is placed into three different functions calls: `epoll_create`, `epoll_ctl` and `epoll_wait`. The `epoll_ctl` call allows to modify the monitored set of file descriptors, which ensures that all registrations do not have be passed to the `epoll_wait` every time again. Only a change in an event interest requires a call to `epoll_ctl`. The `epoll()` call has a complexity of O(1) with respect to number of monitored file descriptors.	Same as `poll`. In addition, the system allows to specify the `EPOLLONESHOT` flag, which will remove a file descriptor from the selector as soon one event fires. Finally, edge level triggering for event event notification can be enabled by the `EPOLLET` flag, more about this option in the next paragraph.

BSD `kqueue()` [31]	Provides a generic method of notifying the user when an event happens or a condition holds. `kqueue()` has a complexity of O(1) with respect to number of monitored file descriptors.	In contrast to the previous event providers `kqueue()` is able to monitor other event sources as well. For instance, it is able to monitor for file descriptors (VNodes and socket sources) events, processes events, signals events and multiple timer events. With respect to sockets it supports monitoring of the following two events: **EVFILT_READ** Monitor for changes in `listen()` backlog, or if incoming data is present for normal sockets. Also fires if there is an IO error for the read direction of the socket. **EVFILT_WRITE** Fires whenever data can be written to the output buffer. Also fires if the write part of the connection is shutdown.

Java new IO [25]	The `java.nio` package provides the `Selector` class which allows to poll by the methods `select()`, `select(long)` and `selectNow()`.	Subclasses of `SelectableChannel` can be monitored by the selector by the `register()` method which creates a `SelectionKey` object. This object allows to set the interest for the following events: **OP_ACCEPT** A `ServerSocketChannel` is ready to accept some clients. **OP_CONNECT** The channel is ready to be finalized by the `finishConnect()` method. **OP_READ** Some read data is available at the socket. **OP_WRITE** Some data can be written to the socket.

As also mentioned in the above table, the standard Unix `poll` and `select` system calls have a linear complexity, while the others have a constant time complexity with respect to the number of connections. This is also shown by the benchmarks displayed in Figure 2.2 and Figure 2.4. These figures, and Figure 2.3 do not show a very significant performance difference between `epoll` and `kqueue` systems calls. Remarkably, Figure 2.2 does show that `epoll` performance in general a little bit better than `queue`, while Figure 2.4 shows the opposite behavior. This is probably caused by differences in operating system versions and underlying implementations of the selector.

Edge vs. Level Triggered

An important difference between selectors is the moment they will inform the application about an event. It seems that all above calls are 'level triggered', except the `epoll` selector. This selector allows the developer to configure edge or level triggering manually. The terms *edge* and *level* triggered are also used in conjunction with CPU interruption [68]. The meaning in this context is exactly the same. A *level triggered* event interest, always informs the application whenever an event is 'active'. For instance, if a file descriptor has some data that is ready to read, it will be selected as long as the read buffer contains data. Only if the read buffer becomes completely empty it will stop firing the read event. In contrast, an *egde triggered* registration will only inform the application on an event transition. This means that the read event will only fire once in our previous example, and only if the buffers becomes completely empty first *and*

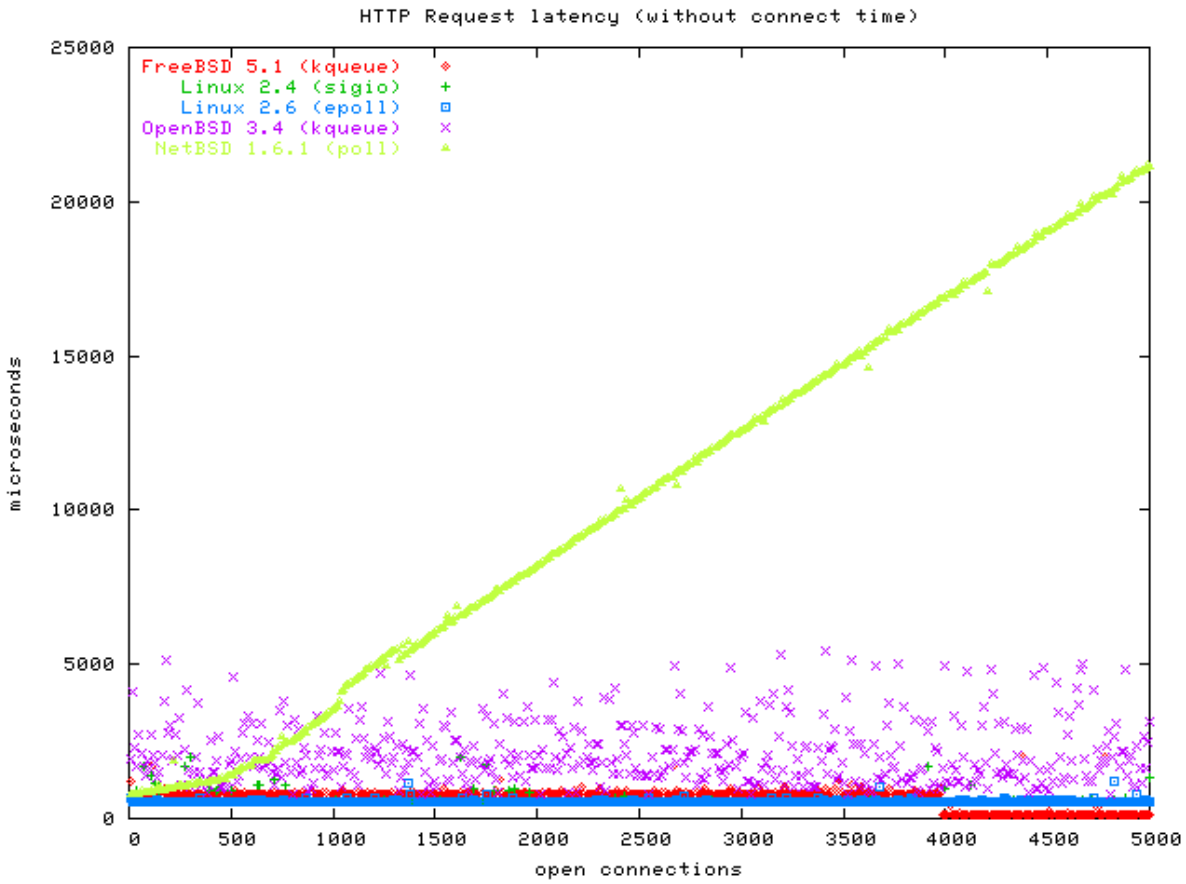


Figure 2.2.: The selector *kqueue*, *sigio*, *epoll* and *poll* [72]. All selectors except the *poll* selector have a constant time complexity with respect to the number of file descriptors, instead, *poll* has a linear complexity.

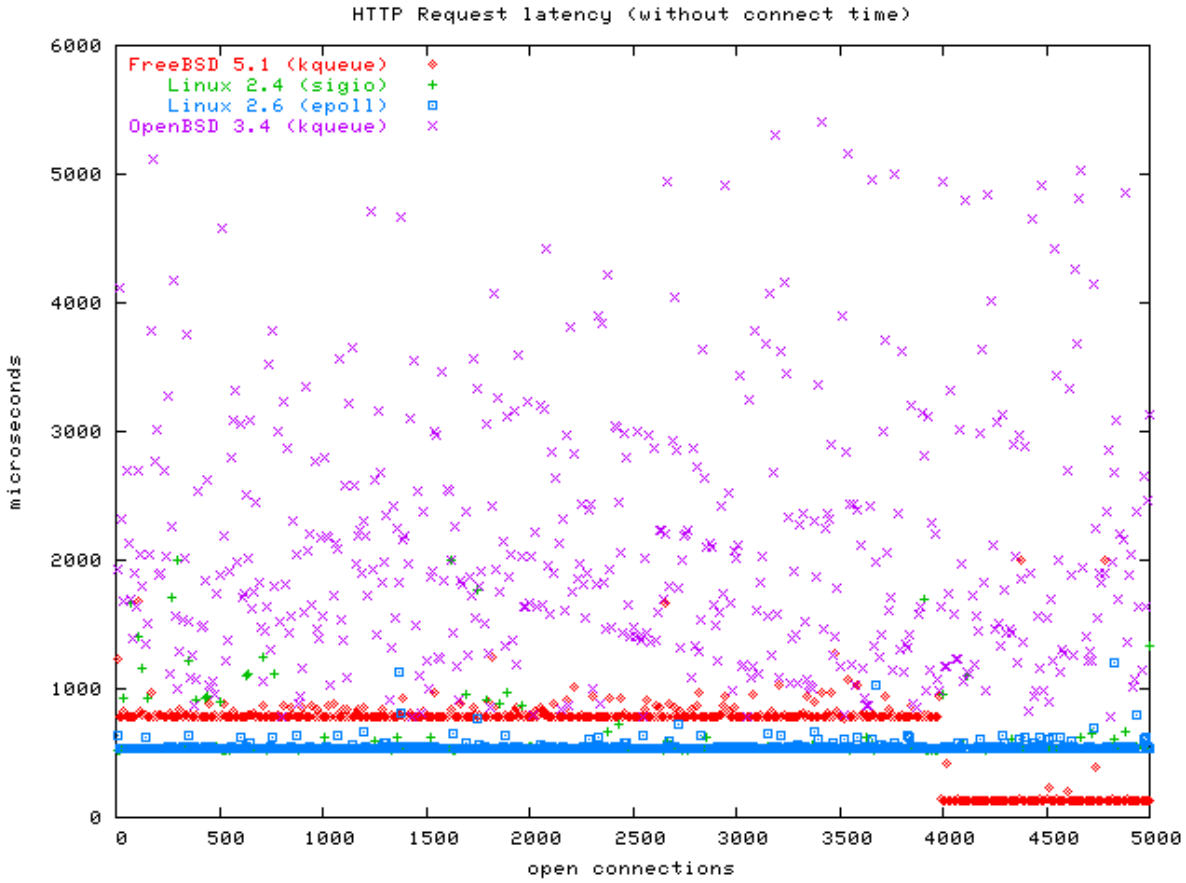


Figure 2.3.: Same as Figure 2.2 but excluding the *poll* system call [72].

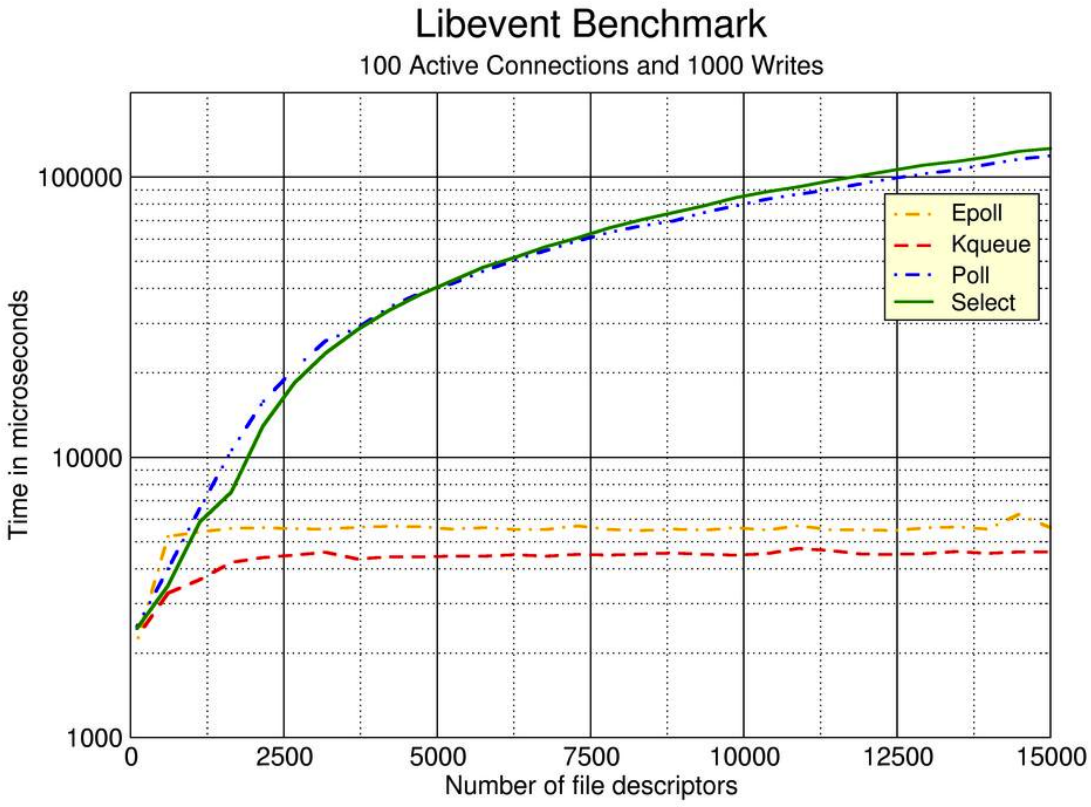


Figure 2.4.: Benchmark of *epoll*, *kqueue*, *poll* and *select* system call [33].

contains new data again, it will fire a new event. In other words, the events are only fired at a rising or decreasing edge.

Some discussions on the Internet denote that edge triggering offers a slightly better performance. However, during the research not any significant proof was found. Also, the manual does not state anything about the performance with respect to edge versus level triggered events. The difference between the two paradigms should especially be found in the way of programming. Edge triggered systems are a little bit harder to program because the system is less fault-tolerant in case an event is accidentally lost since the system will only inform you once. A commonly used trick in combination with edge triggering is to always first try to perform the requested operation (e.g. a read or flush operation), even if the system is not sure if the performed action is ready. Now, in case the event was not ready, the system can itself register for this event at the IO selector. In this case, the system does not have to maintain the actual state of an event. A point of care with this trick is incoming data. In order to ensure that read events will fire again, the system should also always validate if all incoming data is removed from the read buffer.

2.3. Scheduling

An important aspect of the event loop is that it usually schedules the entire (server) application. Everything inside one event loop architecture is usually single threaded, but it is possible that some events are dispatched to separate threads as already explained in subsection 2.2.1.

Owner of the loop In case of such a multithreaded system, it is required that an event loop system can be controlled from this outside world. The main issue here is that because the main loop is in control, the outside world cannot just communicate with this event at any time. In addition, the main event loop is often not programmed in a thread-safe way, which means that, for example, modifying an interest of a file descriptor can only be performed by one thread at the same time. Often, this is the thread of the event loop itself. An alternative would be that all threads (including the main thread) should always acquire a lock before selecting potential events. This will impact the performance in a negative way because the event loop iteration is, as already mentioned, the most important execution cycle in the system.

For this reason, it is a good decision to make the event loop thread the real owner of the loop controls, and to agree that only the event loop is allowed to modify its owned properties. Now, if there are multiple threads inside the system, they should still have to acquire a lock to ensure that only one external thread gains access to the event loop, but the event loop itself does not have to maintain any locks anymore. This because the event loop can just assume that it is in in control until *one* external thread sets a flag (i.e. a normal property, no lock) that that it wants to execute 'something' in the next iteration.

Waking up Another problem that is a little bit hidden in the above story, is that the event loop does not 'know' at what moment an event from the outside world arrives. At the moment such an external thread will set the interrupt flag, the event loop could still be waiting for an event that will be passed by the operating system. In this case, the event loop, and thus the external thread could wait infinitely before this set *interrupt flag* will be discovered. To solve this problem, the event loop must support a kind of *wakeup signal* to get out this sleep state.

Scheduled tasks Next to handling IO events (and other operating systems events), a well designed event loop should be able to schedule executions before a given deadline. This is the responsibility of the event loop because in case there is only one single thread in the system, the event loop is the only component who actually can perform this task. Of course, it is also possible that there is a separate scheduler thread that does nothing more than just executing functions after a given deadline. This looks more simpler from the event loop perspective, but the opposite is true for the developer code. If there are two separate threads for OS events and scheduled tasks, the developer is again required to not share any (not thread-safe) information between these two threads. Thus, from the developer perspective it is much easier to just reason about one single thread for both IO events *and* scheduled tasks.

2.4. Coupling and Integration of existing projects

A very important, but still untreated aspect, is the integration of event loop architectures inside developed applications. Especially the coupling with the rest of a system plays an important role on how flexible the framework can be used. The coupling, for instance, determines how easy another communication protocol can be added, or how easy the application code can be replaced by another event loop system.

For this reason, the coupling of protocols plays an important role for choosing a framework. This is also shown by the Finagle project of Twitter [13], they state on their home page: 'Finagle provides a rich set of protocol-independent tools'. This statement would suggest that their framework is completely compatible with many existing protocol implementations. However, on their developer documentation is stated that the framework only supports a growing set of protocols [14], and that protocols can manually be extended by deriving the *Codec* trait[2] [12]. This trait is for instance implemented by the `ThriftServerFramedCodec` class, which adds support for the *Apache Thrift* protocol.

This way, every new supported protocol requires a new implementation of the `Codec` class. And simply browsing the package of this Thrift protocol proofs that adding a (simple) protocol already involves 14 classes, of which 9 are directly related to the `Codec` class.

[2] A Scala trait is similar to an abstract class in Java, and adds also support for multiple inheritance.

Existing systems The following six subsections will discuss each a different popular framework. In the end of each section a brief overview of the programming and protocol aspects of the framework is provided. Finally, a summary is given that compares the described frameworks. The discussed frameworks are respectively, Twitter Finagle, JBoss Netty, Node.js, Akka, Twisted and finally Gevent.

2.4.1. Twitter Finagle

This project is already mentioned in the above example. Finagle is built on top of Netty as shown in Figure 2.5, the Netty project is discussed in the next section.

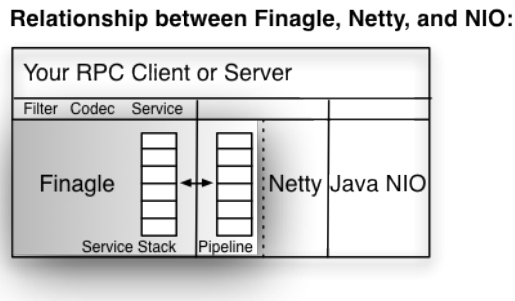


Figure 2.5.: Relationship between Finagle, Netty and Java NIO.

Slogan 'Finagle is a network stack for the JVM that you can use to build asynchronous Remote Procedure Call (RPC) clients and servers in Java, Scala, or any JVM-hosted language. Finagle provides a rich set of protocol-independent tools.'

Supported protocols HTTP, Streaming HTTP (Comet), Thrift, Memcached/Kestrel and SSL/TLS.

Extending protocols Deriving the `Codec` class. The `pipelineFactory()` and `prepareService()` methods should be implemented, the created `ChannelPipeline` object is part of the Netty architecture, and is thus discussed in the next section.

Handler code Finagle is especially created to provide a simpler interface around the Netty framework. The Scala interface to create, for example, a HTTP server is much simpler in this case. Also, extra functionality is added to create HTTP clients with only a few lines of code. Also, the system heavily uses future objects, which will be discussed later on in this document.

Finally, the system allows to bind services (e.g. Thrift services) to an HTTP connection in a really easy way. This last aspect really improves the maintainability of the application code because the business logic is really separated from the protocol/network logic.

Visibility of main loop The main loop of the system is not really visible, this is probably because Netty project also encapsulates the main loop from the outside world. This results into a system where all event loop functionality is nicely hidden from the developer. On the other hand, for example scheduling a timed execution for the main loop becomes difficult because of this reason.

2.4.2. JBoss Netty / Netty.io

JBoss Netty is a popular event-driven and asynchronous Java application framework. A first observation already shows that this project makes clearly a distinction between Protocols and Transports [42]. However, this distinction is not very visible inside the implementation, this is probably caused by the very generic architecture of the used *Interceptor Chain* Pattern [41]. This pattern ensures that all input and output is treated by a chain of handlers, these handlers are derived classes of `ChannelUpstreamHandler` or `ChannelDowntreamHandler`, as shown in Figure 2.6.

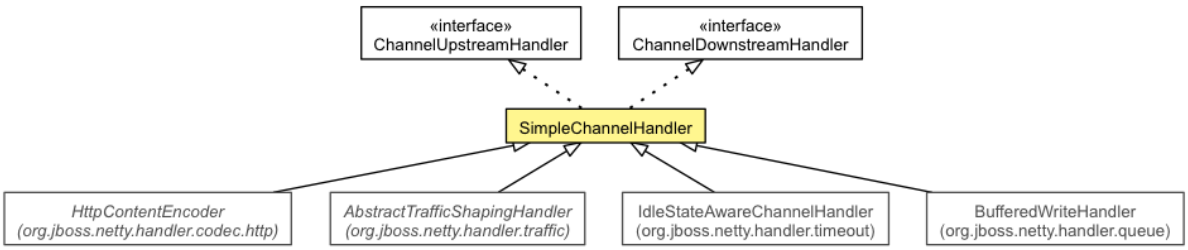


Figure 2.6.: Inheritance relations of the *SimpleChannelHandler* class.

Now it is easily possible to for example add support for the Google Protocol Buffer protocol over a SSL connection. This can be done by respectively chaining the `ProtobufEncoder`, `ProtobufDecoder` and the `SSLHandler` to the `ChannelPipeline`. New protocols can be added by creating a new channel handler and implementing the `handleUpstream()` and/or `handleDownStream()` methods. Statefullness of a handler is achieved by the `setAttachment()` and `getAttachment()` methods of the `ChannelHandlerContext` class. These methods respectively store an `Object` or retrieve a stored `Object`, casting is required to change the object back to the right type. The attachment functionality is required in order to maintain any state, because as mentioned earlier, the programming stack cannot be used since everything is pure event based.

Netty also offers functionality similar to the proactor pattern explained in section 2.2. The `ReplayingDecoder` in the `org.jboss.netty.handler.codec.replay` package [40] allows to encapsulate the decoding process terms of 'complete only if the read of N bytes is finished' instead of 'X bytes are read, where is X is equal to the amount of bytes received'.

Finally, Netty does not offer a real application infrastructure by, for example, adding a handler framework for specific kinds of requests and responses. Netty is a pure IO framework and projects like Finagle offer functionality on top of it.

Slogan 'Netty is *an asynchronous event-driven network application framework* for rapid development of maintainable high performance protocol servers & clients.'

Supported protocols Transports: Socket, datagram and HTTP tunneling. Protocols: HTTP, Websocket, SSL/TLS, Google protocol buffers, ZLib/GZip compression and some textual protocols (e.g. telnet protocol). These protocols can be combined by using the Interceptor Chain design pattern.

Extending protocols Implementing a new `ChannelHandler` class, and chaining it to a `ChannelPipeline` instance. Helper functionality is also included by for instance the `ReplayingDecoder`.

Handler code All IO based, the actual application infrastructure should be build on top of this framework.

Visibility of main loop By default Netty seems to not offer a main loop to the outside world.

2.4.3. Node.js

Compared to the previous two examples, Node.js is a complete application framework with much more functionality than only IO handling [22]. With respect to this IO functionality, Node.js uses the reactor pattern and thus only supports `data`, `end`, `error` and `close` events for readable streams. The developer itself is completely responsible for creating handler code that forwards the data when an event is completed. Extensions like *Connect* provide this functionality for incoming HTTP requests.

The handler code is usually constructed by *callbacks* and *closures*. And as with most javascript code, it does not really use classes to create new handlers, only a simple callback registration is enough to proceed. This gives a lot of freedom to the developer, but the freedom of loosely coupled callback code can easily result in very unmaintainable code.

Slogan 'Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.'

Supported protocols HTTP, SSL/TLS, datagram, ZLib and GZip. RabbitMQ, AMQP, MsgPack and Websockets are supported by extensions.

Extending protocols Protocols can be added, but this is usually performed by just wrapping the low level socket primitives in Javascript (or in some cases in native C++). Node.js does not seem to provide a default architecture for protocol design.

Handler code All based on *callbacks* and *closure* functions.

Visibility of main loop The main loop (i.e. the chrome V8 engine) is hidden in the entire system, but all javascript calls register events. For instance, the *setTimeout* call will just register a callback in the V8 engine, which will be executed by the main loop as soon as the deadline is exceeded.

2.4.4. Akka

Akka [20] offers a quite modular design compared to the previous projects, and is much more designed from the developer perspective than for example the Netty project that more or less emerges from the underlaying system calls. However, it should also be mentioned, that all these individual components are specifically designed for Akka. And it seems that these cannot directly be integrated in other systems as well.

Akka itself is especially designed around the Actor programming paradigm. An Actor is a software component that is able to receive incoming and send outgoing messages from and to other actors. The asynchronous programming model is actually also forced by this architecture. IO communication with the event loop is actually performed by the use of Scala operators and/or normal method calls to remote actors. In addition, it is also possible to interact directly with the main loop, which is split up in two parts in Akka. The *Event Bus* [10] component is used for sending and subscribing to messages, and the *Scheduler* [48] component allows to schedule computations after a given timeout or repeatedly at a given interval.

Because Akka provides a high level architecture, the real message and transport protocols are not visible inside the business logic. Protocols are actually just configured by the Actor configuration. In this file, it is possible to define *Serializers* [50]. A serializer is responsible for implementing the `toBinary` and `fromBinary` methods to respectively marshal and unmarshal the objects. In this way, new serializers can just be created by deriving the `Serializer` class [50, Customization section]. An important negative aspect of this approach is that the serialization code is not allowed to know what kind of object will be (de-)serialized. This must always be performed with *dynamic type checking*.

The attitude of Akka to transports is quite the same as to protocols. Transports are also configured inside the configuration file, and the business logic does not care about to which or what system the message should be delivered.

Finally, an important open question is how Akka actually knows when a *complete* message arrives. The `fromBinary` method of the `Serializer` implementation just assumes that the entire message is arrived, as with the previously explained proactor pattern. However, it is not clear how the framework itself actually knows that all bytes are arrived because the `toBinary` method does not include any length information. A possible explanation could be that the underlaying system will piggyback all messages with additional length information, but this cannot be concluded from the manual.

Slogan 'Akka is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM.'

Supported protocols At least Google Protocol Buffers and Java Serialization, and SSL and normal socket communication as transports.

Extending protocols Message protocols can be added by extending the `Serializer` class. It is not explained by the manual how new transports can be added.

Handler code An Actor implementation will receive the incoming messages which can be handled by a Scala `case` matching structure. The actor itself is allows to maintain a state, but a shared state with other Actors is not allowed for thread-safety reasons.

Visibility of main loop Provided by the *Event Bus* and *Scheduler* components. The role of the event bus is nicely encapsulated for communication between actors (including the role of message boxes in between).

2.4.5. Twisted

Twisted is an open source networking engine written in the Python programming language [19]. Compared to Akka, Twisted offers a much more low level approach to network programming. The architecture is based on the reactor pattern and allows handling of the `dataReceived`, `connectionMade` and `connectionLost` events. This functionality is offered by the `Protocol` class [54]. The `LineReceiver` is a specialization of this class and replaces the `dataReceived` method by the more developer friendly `lineReceived`. In other words, these two methods show again the difference between the reactor and proactor patterns.

Twisted does not really support message protocols, they push this responsibility to the developer. However, they support `Deferred` objects, which is their implementation of future objects.

Slogan 'Twisted is an event-driven networking engine written in Python and licensed under the open source MIT license.'

Supported protocols Low level streams and datagram protocols (including multicast support), HTTP including SSL/TLS support.

Extending protocols Deriving the `Protocol` and `Factory` classes in the `twisted.internet` package.

Handler code Only event based, the developer is responsible for separating business and network logic. Tough, the HTTP or LineReceiver packages allow higher level of abstractions.

Visibility of main loop The `Reactor` class represents the main loop.

2.4.6. Gevent

Like Twisted in the previous section, Gevent is also a python event-based networking library [21]. In contrast to Twisted, Gevent claims to be more developer friendly and should provide the same functionality as Twisted but with less code complexity. The most important reason for this is that Gevent uses *Greenlet*s for executing event handlers. Greenlet is a *micro-threading / fiber* implementation that allows to do co-operative multitasking [18, 58]. Co-operative threads are similar to normal threads, except that they only perform a context switch whenever the active thread releases itself. This paradigm is also called *non-preemptive scheduling*.

Non-preemptive scheduling allows Gevent to provide synchronous API's for asynchronous tasks, this because the state of the programming stack can be preserved by the Greenlet. And a developer friendly aspect about this is that Gevent allows the use all normal synchronous networking API's. This is performed by *monkey patching*, which is a way of modifying code at run-time that is only possible with scripting languages [60].

Slogan 'gevent is a coroutine-based Python networking library that uses greenlet to provide a high-level synchronous API on top of the libevent event loop.'

Supported protocols Normal sockets and SSL are supported, in fact, all blocking libraries in python that are monkey patched libraries are supported.

Extending protocols 'Blocking reads' can just be performed from the standard patched socket library.

Handler code No callbacks are required, just blocking code can be used which really simplifies programming. The system does not force the developer to a predefined application infrastructure.

Visibility of main loop Completely implicit and integrated in the Greenlet architecture. The `Greenlet` class can be used to schedule other computations from the main thread and to control the co-operative threads (e.g. starting, sleeping, joining and killing).

2.4.7. Summary

The previous subsections showed five popular event loop libraries. These are all discussed from the network and programming perspective. This section is intended to give a brief summary and comparison of the discussed libraries.

Both Twisted (Python) and Netty (Java) provide quite similar functionality. Both can be seen as very thin wrappers around the low level operating system primitives, and they more or less forward this functionality directly to the API user. These wrappers are implemented in line with the Reactor pattern, but both libraries also support some proactor functionality on top if this. For instance, the `lineReceived` and `ReplayingDecoder` functionalities of respectively Twisted and Netty.

In the case a new message protocol should be designed, both libraries require a new implementation of an event handler class (i.e. `ChannelHandler` in Netty and `Protocol` in Twisted). Such a new handler implementation might look quite simple at first hand, but recall that these handlers are not allowed to maintain any state on the programming stack. And in addition, blocking parse methods of a codec package can also not be used because the incoming frames will arrive with unknown length.

Node.js, Akka and Finagle provide much more functionality compared to these two projects. Node.js and Akka can be seen as complete application infrastructures, while Finagle has a little bit more lightweight design. Finagle also has the same protocol structure as Netty, so the above story applies again.

From the network development perspective, Node.js is actually the most complex solution for a developer. There is no protocol design part, and the developer will stand alone if he wants to add support for a new protocol. The only functionality Node.js provides is a bunch of standard asynchronous (i.e. all with callbacks) Javascript calls without any design on top. This results into that any advanced networking code will become very complex because, for example, the *readable streams* only supports the registration of an incoming read event. This is actually at the same programming level as Netty and Twisted provide, while these libraries support (or easily could support) more modern future object functionality as well.

Akka surely offers the most features of all projects (e.g. fault-tolerance, supervision, routing, etc), and also really provides a well designed and described framework architecture to the developer. From the IO perspective, Akka does not really describe how to extend the low level *remote* part of their framework. This is also caused by the use of the Actor pattern architecture which enforces that all communication is message based, that is in Akka solved by serialized Java/Scala objects. In other words, Akka is not really designed to, for example, implement a new video service protocol, but more for the simple message communication between different (remote) nodes.

The last discussed project actually is completely different compared to the other projects with respect to programming. Gevent uses coroutines for scheduling just normal blocking code. In theory, this allows the developer to just use all blocking API's provided by the standard Python library. Now, in contrast to all the other projects, the developer can just write blocking code without worrying about underlying asynchronous communication structures.

3. (Non-) blocking programming analysis

Chapter 2 already mentioned that there is usually a high coupling between the event loop framework and the actual communication code. The intention of this chapter is to provide an analysis about this from the programming perspective of the framework developer or user.

This chapter is divided into two different sections. This distinction is made because the communication logic of an application is often split up in (at least) two separate levels: The pure IO interaction level (i.e. how are the messages written to and read from the network), and the application logic itself that controls this lower IO level.

The first section will discuss the requirements from the low level IO perspective. It will analyze a few popular serialization projects, and finally discuss how programming languages lack in the support for non-blocking IO operations. The second section will treat asynchronous programming with respect to the higher level application logic, and does this by two different case studies. These two case studies are discussed for each of the four explained approaches. Finally, a summary with the most important benefits and liabilities of all approaches is given.

3.1. IO Programming and Design

In section 2.4 is shown that five of the six discussed systems require an entire new class implementation varying from a complete new non-blocking parser structure to a simple read call in case of Akka.

Gevent takes a different approach which is less verbose. This project allows the developer to use the normal blocking API's and thus to totally forget about the underlaying non-blocking behavior. The Gevent library does this by wrapping all blocking network calls of the standard python API and this allows the developer to just write pure linear code. In this case, the extension of a new protocol can be as simple as a modification of one programming statement. Though, in a well designed environment this is still encapsulated within a class, so the application stays more adaptable and maintainable. The intention of this section is to elaborate on this discussion about blocking and non-blocking programming of network read and write actions.

3.1.1. Parsers and serializers

Before discussing the real programming differences between blocking and non-blocking IO, it is important to see first what kind of data is actually transferred. In a well designed system, the actual communication interface is as generic as possible in order to keep this part of the system as generic as possible. But from the programming perspective, the parsing and writing part is always quite similar. This is shown by the table below, which shows five popular serialization formats that can be used to transform local objects into transmittable byte sequences. The table shows the marshal and unmarshal code for one supported language per serialization project.

Protocol	**Marshal**	**Unmarshal**
JSON in HTML5 [28]	`var messageData = JSON.stringify(jsObj);`	`var jsObj = JSON.parse(messageData);`
BSON (ebson Java library) [7]	`BsonDocuments.writeTo(buf, document);`	`BsonDocument newDocument = BsonDocuments.readFrom(buf);`
Google Protocol Buffers (C++) [45]	`obj.SerializeToString(&str);` `obj.SerializeToOstream(&out);`	`obj.ParseFromString(&str);` `obj.ParseFromIstream(&in);`
Apache Avro (C++) [1]	`avro::serialize(writer, value);`	`avro::parse(reader, value);`
MessagePack (Java) [36]	`byte[] bytes = msgpack.write(obj);`	`MyClass obj = msgpack.read( bytes, MyClass.class);`

A first observation directly shows that none of the projects supports any kind of non-blocking functionality. All projects just translate a string of known length or use language-standard blocking stream for reading the data from. This actually proofs that these programming structures are completely incompatible with the listed asynchronous projects in the previous chapter. Only Gevent would be able to call one of the the above `parse` methods without any additional logic.

A solution for this problem would be to read as much bytes as the upcoming message payload exists of, and to create a temporal byte-stream that can be used when all data has arrived. Thereafter, this byte-stream (e.g. `ByteArrayInputStream` in Java) can be passed to the blocking parsing API's. However, this method has two important liabilities. The first one is that not all protocols can be supported because the message length must be known before any data is passed to the parser. This is easily possible in case the protocol is designed by the developer itself, because it is possible to prepend a fixed length header to the message, or to specify a special delimiter character that indicates the end of a message.

However, if the protocol just exists of a sequence of raw bytes that cannot be extended, it is just impossible to use a blocking parser implementation. In this case, the

entire parser implementation should be modified to support non-blocking streams, i.e. the parser state must be preserved during multiple subsequent read events. This highly complicates the parser implementation. The second liability is related to system performance. This because either the fixed length header approach or delimiting character approach require both additional copying or additional validation of data.

Another common solution to still support this type of protocols is to rely on the fact that the `ByteArrayInputStream` will through a specific kind of exception whenever a read statement is performed on an empty buffer. However, this cannot be a normal `IOException` that is also thrown if the connection is broken or another error happened. And, the most important liability is that this approach requires that the used parser implementation does transform all raised exceptions into specific 'parser exceptions'. The framework should always be able to distinguish a normal runtime exception with a 'cannot continue' exception.

3.1.2. Standard API support for non-blocking IO

The reason why such a serialization project does not support any non-blocking behavior is probably caused by this extra involved complexity. In addition, currently it seems that there are no programming languages that do support non-blocking streams in their standard API's. For instance, Java started to support non-blocking IO by introducing their `NIO` package, but they did not modify their *InputStream* interface to become compatible with non-blocking sockets as well. They could have done this by modifying the `read()` method of this class so it could also return if there would be temporarily no more data available in the buffer. This could be implemented in the same way as the end-of-stream is indicated for the `read` method, i.e. by returning -2 instead.

It might be clear that this solution would directly make every existing library incompatible with the new *InputStream* implementation. And this would also cause that developers from now on should worry about the case there is temporary no data available. In other words, *all* code must anticipate if the read buffer becomes empty, and thus should maintain the internal parsed state. Another solution would be to rollback all the performed read operations, and put the read data back into the buffer so it can be read again if there becomes more data available. For instance, C++ supports this functionality by the `unget` and `putback` methods [62, p. 683]. So, this solution seems to be quite feasible at first hand, however, again think about the new issues that would arise. Now, the developer always has to set a marker before actual read operations can be performed. In this way, a rollback method could move the read pointer back to position of the marker. So far it would be feasible. However, the biggest problem is that the system has to maintain a buffer that is as large as the largest sequence of bytes that the system expects to read. This number is usually not known in advance, and should thus be guessed, which is risky.

Blocking IO libraries are exactly designed to hide all this inconvenient behavior for the developer. Finally, we can conclude that with blocking IO the developer is fully in control, while with non-blocking IO the event loop or operating system is in control.

3.2. Asynchronous application programming

The previous section treated discussion about non-blocking versus blocking IO. The provided viewpoint point is from the message parse and write level. In contrast, this section will elaborate on the same discussion, but now from the application logic view point. In order words, the question now is: *How do the differences between blocking and non-blocking IO operations affect the developer experience when programming the logic 'above' the message parsing layer?*

This question will be answered by two different case studies that are introduced in the following section. Both studies, combined with an abstract description, are discussed for each approach. The approaches *Blocking*, *Callback driven*, *Future objects* and *Linear code provided by coroutines* can respectively be found after subsection 3.2.1. Finally, the section concludes with a summary of all approaches with respect to *Threading*, *Top-down code*, *Top-down execution*, *Parallelization of IO* and *Performance*.

3.2.1. Case studies

The upcoming subsections use two recurring examples to illustrate the different ways of programming. The first example is about a generic system that will dispatch tasks to a separate layer in a reliable way. The goal of this example is to illustrate the programming difficulties that will arise when a non-blocking event based structure is used across one single connection. The intention of the second example is to display the similar difficulties that occur when maintaining multiple connections in the same event based environment.

Task monitor This case study is about a system that contains two layers. The *TaskMonitor* layer and the *TaskLayer* itself. Both layers may exists of multiple processes, which might be distributed among multiple physical servers. A *TaskMonitor* instance is responsible to dispatch the tasks to an instance inside the *TaskLayer*, and it should ensure that the task is executed without wasting a considerable amount of time in case of a failure in the *TaskLayer*. The actual processing of the tasks might take a long time because a task could require a lot of external communication and/or long computational processing.

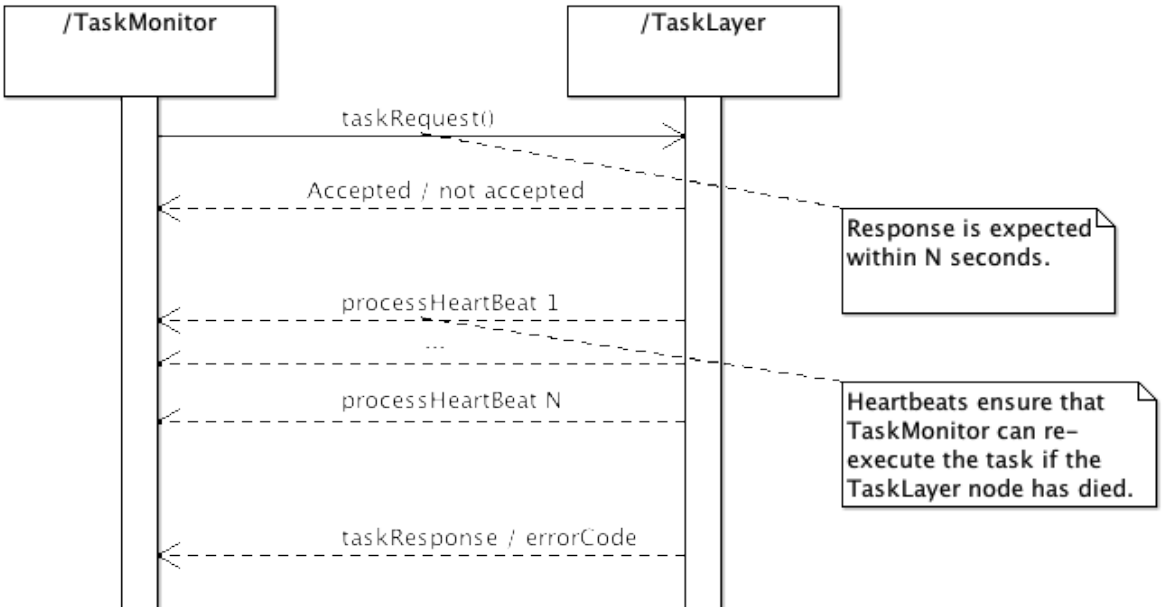
To ensure a good response time of the *TaskMonitor* layer, the system has to perform the execution requests in two phases. The first phase will start to query a node in the *TaskLayer* if it is able to handle the new task or not, e.g. due to load balancing issues or an internal failure of the *TaskLayer*. The load balancing itself is not inside the scope of this example. This step is followed by the second phase which is the actual execution of the job and should reply the result back to the *TaskMonitor*. The separation of these two phases will ensure that the system does not have to wait for the expected calculation time in case the execution cannot be handled. A heartbeat implementation is used to detect potential failures during the execution of the task.

To summarize, the following steps should be taken in order to execute a new task:

1. Connect to a node in the *TaskLayer*.
2. Send the execution request for the task, including the required parameters for the *TaskLayer* instance.
3. Within a configurable amount of time, the system must reply if the task can be accepted or not. If this message is not received for some reason, it is assumed that the system could not handle the task.
4. After the execution of the task is accepted, the *TaskLayer* will execute the task and periodically reply a heartbeat message so the *TaskMonitor* knows that the task is still pending.
5. If the processing of the task has been finished, the response code is send to the *TaskMonitor* and the connection is closed.

The above steps are also displayed in the sequence diagram of Figure 3.1.

Figure 3.1.: Sequence diagram of the task monitor case study.

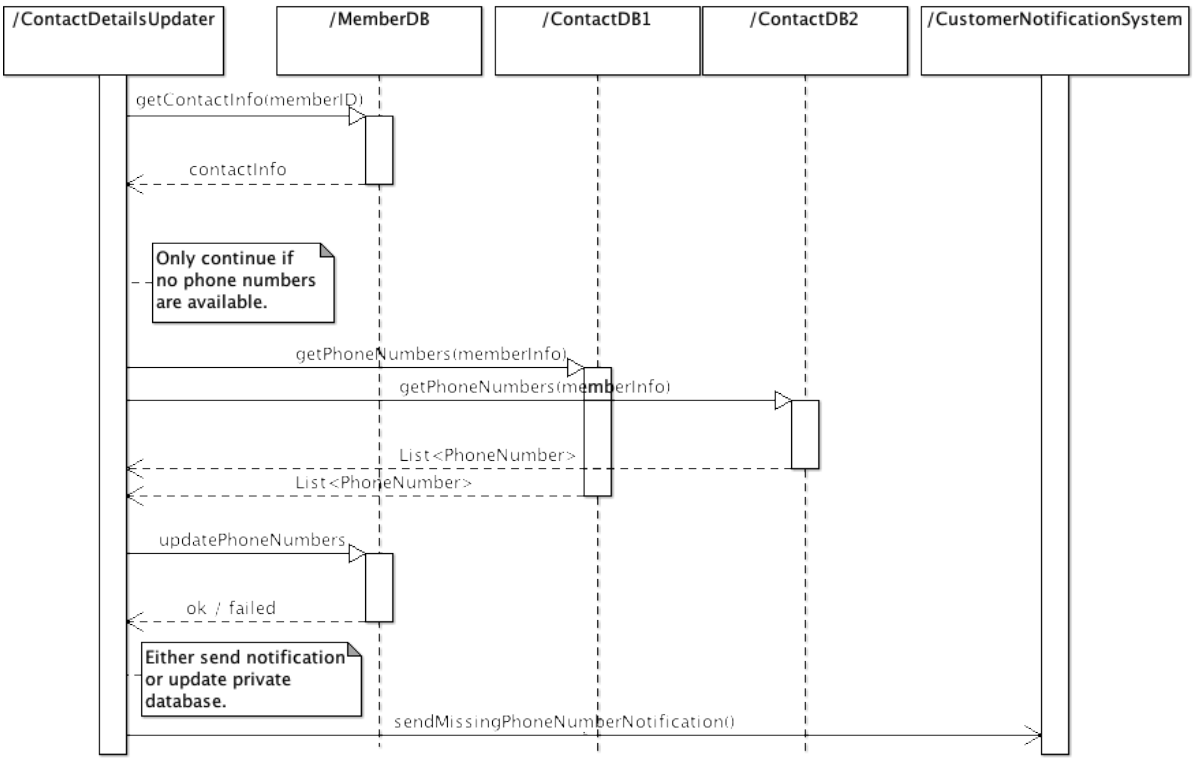


Contact information collector The second case study is about an imaginary system that will collect all contact information of a connected user. The precise information that must be collected depends on a set of given rules that is displayed below. These rules will result into conditions based on the incoming information inside the implemented code, which is usually the case in real projects (e.g. inside the business layer of a three tier architecture). The used message protocol itself is not important for this case study, but it should be clear that the protocol is based on a request-response pattern as for example HTTP or REST do.

1. First request the private database `MemberDatabase`. If this database returns one or more phone numbers, the process can terminate.
2. Request two external databases for phone numbers of the user (`ContactDB1` and `ContactDB2`). This might be performed in parallel.
3. Finally, store the phone numbers found inside the `MemberDatabase`, or send a notification to the member that he should update its contact information as soon as possible. The `CustomerNotificationSystem` is responsible for this part.

The above steps are also displayed in the sequence diagram of Figure 3.2.

Figure 3.2.: Sequence diagram of the collection steps.



3.2.2. Blocking

Blocking code can be seen as the traditional way of programming. The developer calls a function and the function either returns the result of the operation, or throws an exception. This results in code that is understandable because the logic 'flows' from top to bottom.

Task monitor In this case, the use of synchronous code automatically results into a threaded system, because it is required that the *TaskMonitor* component can dispatch multiple tasks at the same time. The application code of the this approach is really

similar to the coroutine approach that will be displayed in Listing 3.7 for the coroutine approach. The only difference is that blocking code would not use the future placeholder at line 6. Instead, it would just execute the normal call, and set the timeout before performing this step.

Contact information collector In Listing 3.1 a self explaining blocking implementation is shown. The readability of this implementation is, as expected, very good. But there are still issues with this blocking approach. First of all, it has to use threads to be able to do multiple requests at the same time. This gives some liabilities like, the `memberDB`, `contactDB1`, `contactDB2`, `notificationSystem` and `logger` are shared across all threads, which means that they should all be implemented thread-safe. This concretely means that all public accessible methods should be synchronized which will give a performance penalty.

In addition, the requests at line 7 and 8 could easily be parallellized. This could improve the response time of the `update()` a lot, especially if the external contact databases are responding slow.

Listing 3.1: Pseudo implementation in Java of the blocking approach. To ensure parallelism between multiple requests it is required that the *update()* is executed by multiple threads.

```
1  public void update(int memberID) {
2      ContactInfo info = memberDB.getContactInfo(memberID);
3      if (info.hasPhoneNumber()) {
4          logger.info("member[" + memberID + "] is already updated");
5      } else {
6          // Phone numbers not present in database, so external lookup:
7          List<PhoneNumber> req1 = contactDB1.getPhoneNumber(info.cid);
8          List<PhoneNumber> req2 = contactDB2.getPhoneNumber(info.cid);
9          if (req1.isEmpty() && req2.isEmpty())
10             notificationSystem.sendMissingPhoneNumberNotification();
11         else if (!memberDB.updatePhoneNumbers(merge(first, sec)))
12             logger.error("Could not update phone numbers at database");
13     }
14 }
```

3.2.3. Callback driven

Traditionally, callbacks are implemented as function pointers in low level languages. The caller of a function specifies the function (or functions) that should be called whenever something is finished or an event happens. Modern languages have adopted this behavior by using references instead of pure pointers. This extension was required to ensure that object oriented languages could use something similar for class methods. A normal function pointer translates to the plain address of the function inside the application memory. A call to a class member function requires a pointer to the object memory as well. Thus, this principle requires theoretical at least two addresses: The function address and the object address. Most modern languages do this *function binding* implicit.

For example, Python just allows to pass a reference to a member function, which is a special type. And in Scala every member function can even be referenced as it were a normal object.

To make the use of (simple) callback functions even more easier, constructs for *anonymous functions* were added to programming languages. Anonymous function are implemented inline inside the calling context and do not have the programming overhead to write down the entire function blueprint. This kind of functions do often have the size of one statement, but it can also contain multiple statements. In some cases, the anonymous functions are also able to capture variables from the calling context, this set of variables is called the *closure*. These captured variables can be seen as additional parameters for the function that is not passed by the caller of the callback, but instead by the creator of the inline function. The C++11 lambda functions display this behavior in the most explicit way by allowing the developer to specify the capture properties between the `[` and `]`-brackets. In this way, the developer is able to specify if the function should be passed by value or by reference.

This last behavior already shows a part of the additional complexity in non-garbage collected environments. A normal function call allows to pass the objects by reference, even if they are created on the programming stack. However, this is not possible if the call to the anonymous function is performed after the stack already has been changed. This last behavior is exactly essential for event loop events, these events always occur *after* the original programming context has changed.

Task monitor The statement that application logic results into a complex state machine when using callbacks is already shown in the Javascript code in Listing 3.2. The two phases mentioned in the introduction are explicitly visible inside the double nested inline functions (i.e. line 4 and line 7). In addition, the state of the heartbeat logic is represented by the single closure variable `timerID`, this timer should only fire as the heartbeat did not arrive within the specified interval (at line 14). If this happens, the entire request is assumed to be failed, and the `executeTask()` function should be re-executed again. This same behavior is required whenever the *TaskLayer* cannot accept the task.

A vigilant reader might already have noticed that the implementation in Listing 3.2 is not correct. Because the application is not allowed to maintain any state, the programming flow for even such a simple application already becomes quite complex. If we have a look at the heartbeat logic of this application, we see that it is controlled in three places, at line 7, 15 and 18. The first one creates the timer, the second one resets it in case an heartbeat arrives and the last one destroys it because the response arrived. So, this functionality can be probably be considered as correct. The hidden bug is that we have forgotten two important state transitions. In case the task execution is considered as 'failed' (i.e. in lines 10 and 26), the heartbeat timer should also be destroyed. In this incorrect implementation, the heartbeat timer would re-execute the `executeTask()` infinitely.

This simple example demonstrates that asynchronous communication combined with

callbacks usually results into very complex and unmaintainable code. Of course, a well designed application would in this case totally encapsulate the heartbeat behavior, but even in this case the entire design would still be complex due to the callback structure. This because programmers, probably like any other human beings, tend to write (code) in the order they reason about the steps that must be taken. Callbacks require that all state transitions are designed (and tested) with care, and can surely not be written down in the way the developer 'thinks'.

Listing 3.2: *Incorrect* pseudo implementation in Javascript using inline functions with closure variables.

```
function executeTask(taskParams, responseCb)
{
  var connection = executorScheduler.newConnection();
  connection.sendTaskRequest(taskParams, function(accepted) {
    if (accepted)
    {
      var timerID = setPeriodicTimer(heartbeatMS, function() {
        logger.warning("Active connection with task layer " + connection +
                       " seems to be broken, heartbeat failure.\n");
        executorTask(taskParams, responseCb)
      });

      connection.onResponse(function(response) {
        if (response.type == HEARTBEAT_TYPE_ID)
          resetPeriodicTimer(timerID);
        else if (response.type == RESPONSE_TYPE_ID)
        {
          destroyPeriodicTimer(timerID);
          responseCb(response.content);
        }
      });
    }
    else
    {
      logger.warning("Failed to accept task at " + connection + "\n");
      executeTask(taskParams, responseCb);
    }
  });
}
```

Contact information collector The two requests to the external databases can either be implemented in sequential or parallel fashion, both implementations are respectively shown in Listing 3.3 and Listing 3.4. The sequential implementation makes use of a nested closure function, so it is clear that the first request is finished before the other finishes. So, in this case it is not required to join both requests as is done for the parallel implementation. The join is only performed when `firstArrived` contains the value of the first completed request at line 13.

It might be clear that both implementations have a very low code quality. Moreover, imagine that more than two requests should be joined, possibly even with different

response types. In this case, the code quality would be even lower because of the high number of indentations and global variables. Finally, please also observe that the current parallel implementation cannot even be executed by multiple callers, because the functions share the same `firstArrived` variable.

Listing 3.3: Sequential implementation of the contact collector by using a nested closure function.

```
function update(memberID) {
  memberDB.getContactInfo(memberID, function(info) {
    if (info.hasPhoneNumber()) {
      logger.info("member[" + memberID + "] is already updated");
    } else {
      contactDB1.getPhoneNumber(info.cid, function(req1) {
        contactDB2.getPhoneNumber(info.cid, function(req2) {
          if (req1.isEmpty() && req2.isEmpty())
            notificationSystem.sendMissingPhoneNumberNotification();
          else
            memberDB.updatePhoneNumbers(merge(first, sec)), function(
                success) {
              if (!success)
                logger.error("Could not update phone numbers at database")
                    ;
            });
        });
      });
    }
  });
}
```

Listing 3.4: Parallel implementation of the contact collector by using a global variable.

```
function handleJoinedRequests(req1, req2) {
  if (req1.isEmpty() && req2.isEmpty())
    notificationSystem.sendMissingPhoneNumberNotification();
  else
    memberDB.updatePhoneNumbers(merge(first, sec)), function(success) {
      if (!success)
        logger.error("Could not update phone numbers at database");
    });
});

var firstArrived = null;
function handleRequest(req) {
  if (firstArrived === null)
    firstArrived = req;
  else
    handleJoinedRequests(firstArrived, req);
}

function update(memberID) {
  memberDB.getContactInfo(memberID, function(info) {
    if (info.hasPhoneNumber()) {
```

```
22          logger.info("member[" + memberID + "] is already updated");
23        } else {
24          contactDB1.getPhoneNumber(info.cid, handleRequest);
25          contactDB2.getPhoneNumber(info.cid, handleRequest);
26        }
27      });
28    }
```

3.2.4. Future objects

A part of the issues that we have seen with the use of callbacks can be solved by using future objects instead. A *future* (or promise-, deferred-, delayed-) *object* is a placeholder in the sense of a normal object that represents a futuristic value that *is* or *is not yet* arrived. Please note that the names future and promise should not be confused with the commonly used *promise-future* design pattern, where the promise does/will contain the actual promised value and the future is just the external read-only view to this promise.

The idea of the future object pattern is that the developer is able to work with this futuristic value as it were the value that is already arrived. Future classes usually have functionality to attach different kind of handlers. And if these handlers are only used without any additional functionality, the programming principle is exactly the same as the one for callbacks in the previous section. The powerful aspect of future objects arise when they are *composable* or allow *comprehension* with other future's or statements, which is treated in the upcoming two subsections.

Finally, the biggest liability of future objects is that they use chaining of events. This means that the written code is not executed linearly, but in the way the events happen, this might give strange stack-traces in case of exceptions.

Composable Future's

Composable Future's are based on the following logical theorem:

$$future(A) \odot future(B) == future(A \odot B)$$

Where A and B represent the values that are received, and $\odot$ represents *any* operator that can operate on A and B. Implementations usually just register a completion event at both future's, and apply the operator when both future's are finished. This principle works very well for operators. However, it would also be handy to be able to call a function with the futuristic value. This cannot be performed by passing the plain future to these functions because the legacy function is not aware about future objects, instead it just expects the value it self. This can be solved by calling this function in a transformation event, and ensure that the return value is just a future itself.

Future comprehension

Future comprehension is very similar to composable future's, but now external language constructs can be used to combine the result of multiple future's. In other words, the

future object does not allow to forward all operators, but the developer should, for example, use *for*-comprehension to combine future's. This is for example something that the Akka project does as shown in Listing 3.5 [15]. Because a normal filter can be added to the comprehension, the developer does not have to construct any visible transformation functions outside the comprehension.

Listing 3.5: Combining future's by using scala for-comprehension in Akka.

```
val f = for {
  a <- Future(10 / 2) // 10 / 2 = 5
  b <- Future(a + 1) //  5 + 1 = 6
  c <- Future(a - 1) //  5 - 1 = 4
  if c > 3 // Future.filter
} yield b * c //  6 * 4 = 24
```

Case studies

Task monitor Plain future objects are not perfectly suitable for the task monitor example because the heartbeats are repeated instead of one future result. In most systems this is not really a problem because the heartbeat implementation will be encapsulated by a lower communication layer. However, in this imaginary case study the heartbeat logic is part of the business logic itself, and these repeated incoming messages cannot be handled by future objects. Though, it is of course possible to use a future object for the expected responses, but this is already shown in the upcoming subsection 3.2.5.

Contact information collector The approach with future objects in the Scala language is shown in Listing 3.6. First of all a remark should be made that there are probably dozens of different ways to combine the result of `contactDB1` and `contactDB2`, in this case it performed by using `for`-comprehension. Besides this comprehension, the programming structure is quite similar to the callback approach explained in subsection 3.2.3. This is because the code in Listing 3.6 also makes use of the callbacks, but for the completion events itself. In this case similar issues arise as well, but now combining the results of a parallel request is much easier.

Listing 3.6: Pseudo implementation in Scala using *for*-comprehensions.

```
def update(memberID: Int) {
  val infoFuture = memberDB.getContactInfo(memberID)
  infoFuture.onComplete (info => {
    if (info.hasPhoneNumber()) {
      logger.info("member[" + memberID + "] is already updated")
    } else {
      // Phone numbers not present in database, so external lookup:
      val combined = for {
        first <- contactDB1.getPhoneNumber(info.cid)
        second <- contactDB2.getPhoneNumber(info.cid)
      } yield mergePhones(first, second)
      // Handle the result as soons as both requests are finished
      combined.onComplete(result => {
```

```
14          if (result.isEmpty())
15            notificationSystem.sendMissingPhoneNumberNotification()
16          else
17            memberDB.updatePhoneNumbers(result).onComplete(succes => {
18              if (!success)
19                logger.error("Could not update phone numbers at database")
20            })
21        })
22      }
23    })
24
25  }
```

3.2.5. Linear code provided by coroutines

Some programming languages provide high level coroutine functionality. A coroutine is a function that has multiple entry points for suspending and resuming the computation by maintaining the current programing state. Coroutines can be used to create a non-preemptive (i.e. cooperative) scheduling framework. In this way, the developer is able to write top-down code and just use the programming stack to maintain the application state. This way, all the issues that emerged from the callback method are solved. The nice aspect of coroutines is that they are much more lightweight than normal threads. This because it only switches the programming stack whenever the system really cannot continue (i.e. because it is waiting for some event), or when the developer really wants to reschedule (e.g. if it wants to schedule other operations if the processing of the current tasks might take a long time).

An example of a coroutine project is the Python Greenlets project mentioned in subsection 2.4.6. This project is really implemented as a package itself, and it does not use any 'language-extensions' to achieve this task. This is also not required because the Python programming language can be easily extended in an implicit way for these purposes, as the Stackless project also did [23]. However, some languages also tried to more or less 'embed' the coroutine functionality inside the language. Two examples of such languages are C# and Scala, which are treated in the following two paragraphs.

C# async The C# programming language adds coroutine support by the `async` keyword, which can be placed in front of a method definition, this marker will allow such a method to suspend at the moment an `await` call is executed [38, 39]. At that moment, the caller of the `async` method will just continue with processing, the value returned is a generic `Task` object and can just be used as it were a future object treated in the previous section. C# provides helper methods for these `Task` classes such as `CopyToAsync`, `ReadAsync`, and `WriteAsync` defined in the `System.IO.Stream` class.

Finally it should be noticed that the `async` functions are not dispatched to another thread. The system will only use the caller thread for processing. And as soon as the return value of the `Task` is required in the calling context, the system will again spend computation from the caller thread to the suspended coroutine. So, the developer does

not have to worry about concurrency issues. This is the most important benefit of coroutines above an `await` method of a normal future library with preemptive threads.

Scala continuations The first difference with the C# `async` keyword is that Scala continuations do not require to use the asynchronous functionality with methods only [53]. The `reset` function, which has the same functionality as the `async` keyword in C#, can be used around any statement. This is of course caused by the fact that the Scala language allows to define a partial function within the scope of the `reset` statement, which is again just an anonymous method. Scala also highlights that the `shift` and `reset` functions are just library functions, and not language keywords. However, the question is if this is completely true, because the compiler does require to enable a compiler flag in order to make use of these functions. Though, this might also be caused by the return types of suspended functions instead [70].

The behavior of the `shift` function is also slightly different compared to `await` function in C#. This because Scala did design the continuations, as the entire language, in the functional-programmed way. The `shift` function is always called within the scope of `reset`, and actually *delimits* the coroutine. `Shift` expects a partial function that is actually the callback method, the magic this shift method performs is that it will store the entire context and will continue whenever the callback is called. An important difference with the C# implementation is that the coroutine could actually be called multiple times, this does not matter for the `shift` keyword, from that point it is just a normal callback function.

Finally, the Scala continuations library also supports *suspendable methods*. A suspendable method is a developer friendly way of wrapping a shift call without bothering about the coroutines itself. From the caller perspective, the function call is the same as it were a normal call to a blocking method, this time with the benefit that the caller thread will just continue after the `reset` statement in the caller context. Please keep in mind that this is more developer friendly than the C# implementation because the `Task` wrapper around the return type is not required anymore. Although, a major problem detected later on in the project is that the actual return type is wrapped by a `@CPS`-annotation. This problem discussed in detail in subsection 4.3.1.

Case studies

Task monitor A possible implementation of the task monitor example in C++ is shown in Listing 3.7 and Listing 3.8. The reason why this first example is shown in C++ is because the implementation in chapter 5 does use this language. For now, it can just be assumed that the coroutine implementation is exactly the same as for the Scala language.

A quick scan already shows that this code is much more readable than, for example, the callback approach in Listing 3.2. The programming state is implicitly visible by the statements of execution, as a developer is used to program. And finally, conditions can be handled as it were a old-fashioned blocking application, which was not possible for the future object approach.

The *-operator at line 9 is used to dereference the value of the future, i.e. this operator call will thus block the current coroutine. The suspended coroutine will be scheduled as soon as the right event arrived to wake-up this value. The implementation in Listing 3.7 does not use any callbacks, but in theory this could be perfectly combined for simple programming structures. Moreover, the use of callbacks is actually also advised in the simple cases, because a callback is usually more efficient than a complete context switch. Although, this statement heavily depends on the underlying implementation. But a callback is especially more efficient with respect to memory consumption compared to maintaining an entire programing stack. This because a programming stack should always be allocated with sufficient stack space. This in contrast to callbacks that only have to store the closure variables and reserve space for the actual arguments. Though, a solution for this stack allocation problem is proposed by [66].

Listing 3.7: Pseudo implementation in C++ of a blocking approach with future placeholders to control the asynchronous behavior.

```
1  TaskResponse executeTask(TaskParameters const &params)
2  {
3    while (true)
4    {
5      TaskExecutorConnection connection = executorScheduler.newConnection();
6      Future<bool> accepted = connection.sendTaskRequest(params);
7      // Set accepted to false if connection doesn't respond the deadline
8      accepted.setDeadlineValue(acceptedDeadlineMS, false);
9      if (*accepted)
10     {
11       if (connection.ensureHeartbeats(heartbeatMS))
12         // Task is executed, get response code
13         return connection.getResponse();
14       else
15         logger.warn() << "Active connection with node " << connection
16                       << " seems to be broken, heartbeat failure.\n";
17     }
18     else
19       logger.warn() << "Failed to accept task at " << connection << '\n';
20   }
21 }
```

Listing 3.8: Possible pseudo implementation of the *ensureHeartbeats()* function used in Listing 3.7.

```
1  bool TaskLayerConnection::ensureHeartbeats(chrono::system_clock::duration
     &interval)
2  {
3    while (true)
4    {
5      Future<HeaderMessage> incoming = readHeader();
6      incoming.awaitWithTimeout(interval); // Blocking wait call
7      if (!incoming.valid())
8        return false; // Nothing arrived within interval, thus fail
9      // A heartbeat message (or something else) in arrived...
```

```
10      if (incoming->typeId != HEARTBEAT_TYPE_ID)
11        return true; // Actual payload arrived
12    }
13  }
```

Contact information collector The contact information collector case study actually shows the power of the combination between coroutines and asynchronous code in a perfect way. Line 7 and 8 in Listing 3.9 show how future's can be used to parallelize two requests, while the rest of the code is still written in a pure linear way. The `get` method of the future objects is the same as the `*` operator in the previous example. Finally, lines 13-16 show that for simple operations the callback driven approach can still be used. In this case, this will give the benefit that the system theoretically does not have to perform any context switch anymore after line 9.

Listing 3.9: Pseudo implementation in Scala using coroutines. The *update* method is expected to be suspendable and only called within the scope of a *reset* function.

```
1  def update(memberID: Int) {
2    val infoFuture = memberDB.getContactInfo(memberID).get
3    if (info.hasPhoneNumber()) {
4      logger.info("member[" + memberID + "] is already updated")
5    } else {
6      // Phone numbers not present in database, so parallel lookup:
7      val req1 = contactDB1.getPhoneNumber(info.cid)
8      val req2 = contactDB2.getPhoneNumber(info.cid)
9      combined = mergePhones(req1.get, req2.get)
10     if (result.isEmpty())
11       notificationSystem.sendMissingPhoneNumberNotification()
12     else
13       memberDB.updatePhoneNumbers(combined).onComplete(succes => {
14         if (!success)
15           logger.error("Could not update phone numbers at database")
16       })
17   }
18 }
```

3.2.6. Summary

This subsection will give a short summary of all previously explained approaches with respect to *Threading*, *Top-down code*, *Top-down execution*, *Parallelization of IO* and *Performance*.

Threading

A first observation that should be made is that the traditional blocking approach cannot work without using (non-)preemptive threads. The requirement that multiple requests can be executed at the same time forces a threaded system. All other approaches can

work perfectly fine in a single threaded environment and still having concurrency among different requests.

The coroutine approach makes use of cooperative threads to store the state of the programming stack. But this does not require as much context switching overhead as with preemptive scheduling, because the stack is only switched when a request cannot be completed. This also gives the benefit that traditional locking is not required anymore.

Top-down code

The callback approach will surely give the most difficult and complex implementation if a requests has to maintain any state. This can vary from the internal parser state or a higher level business logic state. The future comprehension approach results for a major part into linear understandable code, with the exception of conditions. As soon as the future value must be handled inside a while loop or if-else statement it will usually again require more complex thinking of the developer.

The coroutine implementation in contrast, gives the best developer experience since it gives the complete freedom of controlling the asynchronous calls and still having linear code.

Top-down execution

Top-down code does not directly mean that the code is executed top-down as well. This is an aspect where the composable future's and future comprehension do fail. These paradigms actually only chain the underlaying callbacks together. During debugging, this will result into complex stack traces because they do not go linear through the application code. Moreover, the use of many anonymous functions also makes debugging more difficult.

Coroutines solve this problem by really maintaining a separate programming stack. Furthermore, thrown exceptions will in this case also go up in the programmed code, while future objects have to catch these into the underlying closure functions.

Parallelization of IO

In a single core environment it is still possible to parallelize outgoing IO requests and thus speedup the entire response time. The examples in the previous sections showed that it is impossible to do this in the blocking approach without starting a new thread. Also, the callback approach becomes really hard to maintain as multiple requests should be joined together. Finally, we have seen that future objects do really improve the ability to parallelize outgoing requests.

Performance

Finally the most difficult topic to discuss without any measurements: Performance. However, under the assumption that a plain function call is always faster than the

creation of a new object or context switch, it can probably be stated that the callback-approach will have the best performance.

This saying, it should directly be mentioned that this does not have to be the case in real world situations. This because the callback-approach heavily relies on the fact that the developer maintains the application state in global variables that are shared among all requests. So, every change in state always requires a change in a global variable, which often requires a special looked up. This last part might for example be performed by doing a `get` operation with some sort of ID inside a global hashmap. These lookups do influence the system performance. Furthermore, because the state is completely explicitly defined by the developer, an incoming message will always require that it should first be determined what the current state is, before anything can be performed. Again, this often results into a global lookup. Moreover, in this case probably also many conditions should be validated before the real processing can start.

The question is how this additional processing time compares to the construction of a combined future object or with a potential context switch. This information can only be gathered by real benchmarks, which will result into a tradeoff between performance and ease of programming.

An important fact is that none of these three approaches do require the use of any mutual exclusions because they are all single threaded. The expectation is that this will have a positive impact on performance.

4. Linear code: Scala Continuations

The previous chapter showed that coroutines are able to improve the implementation complexity for a developer. This chapter will discuss all aspects of the attempt to combine Java NIO and Scala continuations in one system.

This chapter consists of three different sections. First of all an analysis is provided about the system to-be, followed by the discussion of some important implementation details. Finally, the chapter concludes by a results section which will also highlight the most important problem of this appraoch.

4.1. Analysis

This section describes the most important tradeoffs for the system design.

4.1.1. Scala and Java NIO

The decision to implement the system in Scala was rather obvious. First of all, the distributed system department of the university heavily relies on the JVM. Also, the previous chapter showed that the Continuation library is very well designed, especially in combination with the partial functions. In addition, the suspended functions do not require any return type wrapper as the C# language does.

Though, in this case it would still be possible to write a major part of the system in pure Java, and another part in Scala. The decision to not do this was a very simple one as well. Scala simplifies the implementation in many cases compared to Java. The compact syntax and functor functionality ensured that less anonymous classes and interfaces had to be designed, Also, the seamless integration with existing Java API's (e.g. the New-IO library, or collection packages) made the choice for Scala easy.

4.1.2. Continuations

Continuations can be used in two different places: inside the buffer and future classes. This is exactly the same distinction that is made in chapter 3.

Continuated Buffers Continuations are used inside the read buffer to ensure the proactor behavior. If a user requests to read a number of bytes, the continuations will ensure that the method will 'block' until the requested number of bytes is available. The read buffer can do this by storing the continuation inside a variable which represents ans unresolved read operation. It is also decided to let the buffer not maintain a queue of

read statements. This beholds the rest of the system to only read from the buffer in one context at the same time, which is of course the case with normal data streams as well.

The write buffer uses continuations in the opposite way. It ensures that bytes that should be written and do not fit into the output buffer are written whenever there is again a chunk of free space. The internal buffer implementations itself is briefly discussed in subsection 4.2.2.

Blocking Future The three lines of code in Listing 4.1 show exactly how easy continuations can be used. Such a suspendable method ensures that users of the future library can perform blocking operations. At least as long as they perform this inside the scope of `reset`. The future implementation itself will be discussed in subsection 4.2.4.

4.1.3. Single threaded

As already stated in the introduction, the intention of this project is to have only one main loop thread, and no other preemptive scheduled threads inside the event loop itself. This means that the system does not have to use any locks. Though, it does not mean that the system does not have to be implemented in a thread safe way. The details of aspect are discussed in subsection 4.2.1. Finally, how other threads can access the system is explained in subsection 4.2.5.

4.2. Implementation

This section will discuss the most important implementation details of the project.

4.2.1. Event loop

This project uses the previously explained proactor design pattern. The singleton object `mainLoop` is a facade which involves both IO interaction and scheduling of timed executions. These two aspects are divided into two separate components: The `IOLoop` class and `ExecutionQueue` class. The `mainLoop` object maintains a single object of both classes, as shown in Figure 4.1. In addition, the `mainLoop` object also provides an `intercept` method for tread-safety reasons which is explained in subsection 4.2.5.

IO Loop The `IOLoop` class encapsulates the low level IO event selector that is provided by Java new IO. The IO loop provides four important functions to the outside world. First of all the `selectAndExecute` method which will execute the actual polling request to the operating system. This method receives an optional timeout as argument that can be used to set the maximum time that the function is allowed to block if there are no events.

Another important method is the `register` method which receives a `SelectableChannel` object as argument. This method will register a new IO entry point at the selector and attach a new `IOEventListener` object to it. In this way, the system is able to couple

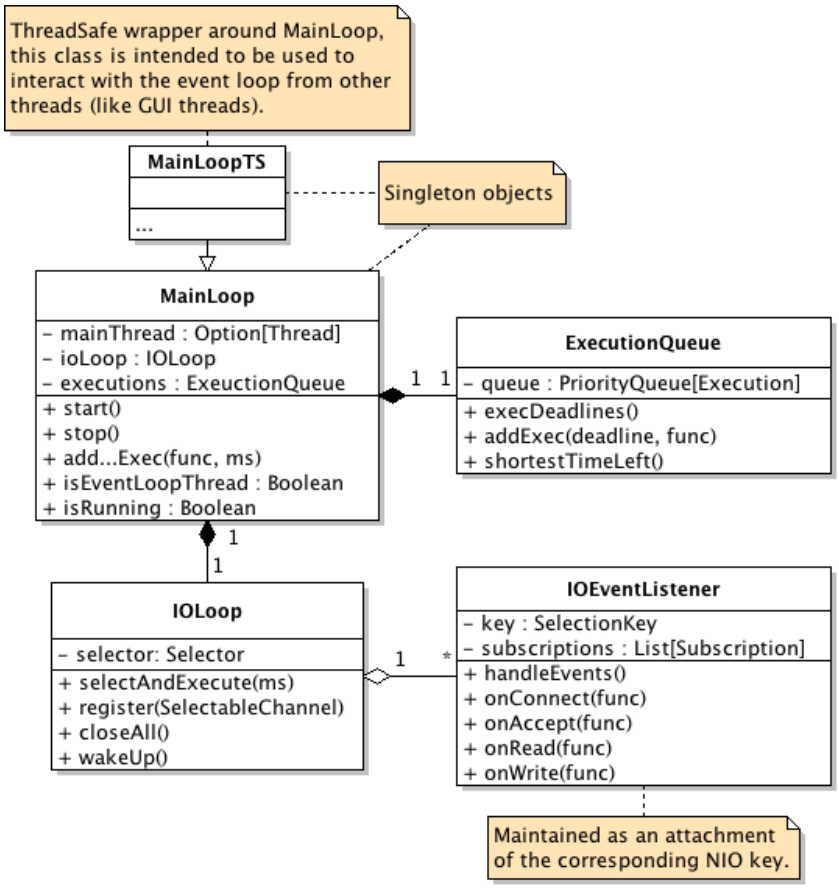


Figure 4.1.: Class diagram of the event loop core classes.

the correct application level event registration to the low level channel object. A channel can theoretically be any kind of channel the new IO package provides, in practice it is often a `ServerSocketChannel` or `SocketChannel`. The `IOEentListener` will maintain all event registrations (i.e. `onRead`, `onWrite`, `onConnect` and `onAccept`) and is able to update the interest mask that is registered. The reason to do this by a callback structure is to maintain a low coupling between the NIO wrapping classes and the actual client and server implementations.

Finally, the class also provides the `wakeup` and `closeAll` methods that respectively interrupt the blocking `select`-call and close all active connections.

Execution Queue The `ExecutionQueue` class provides exactly the functionality that the class name suggests. It maintains a queue of ordered executions that are waiting for their deadline. The `shortestTimeLeft` method indicates how long the `select`-call of `IOLoop` is allowed to block at most.

4.2.2. Java new I/O

As mentioned, the low level IO controls are provided by the NIO package. The behavior of this is encapsulated inside the already discussed `IOEventListener` and another major part is incorporated into the buffer classes. The `ContinuatedReadBuffer` and `ContinuatedWriteBuffer` respectively provide the low level read and write functionality, and both derive from `ContinuatedBuffer` which ensures that buffers can be freed from and to the `BufferPool`. These classes together with the buffer classes are shown in Figure 4.2

An important component of these buffers is the internal `ByteBuffer` object that is part of the NIO package. This `ByteBuffer` object can either be in read or write mode [26]. A buffer should always be in write mode in order to append data to it, whereafter the `flip()` method can be called which will swap the buffer to read mode. Furthermore, the buffer class provides controls to read a sequence of bytes, and to compact or reset the buffer for a new write operation.

4.2.3. Servers and clients

Actually, the `IOLoop` controls two different connections: Server sockets and sockets of normal client connection (incoming or outgoing). Respectively the `NIOServer` and `NIOClient` classes take this responsibility, as shown in Figure 4.2.

The `NIOLengthFieldClient` is a specialization of the `NIOClient` class. It adds support for protocols with four prepended big endian bytes that represent the length of the upcoming message. The abstract method `handleIncomingMessage` and the `sendMessage` method hide this complexity for the rest of the system. This functionality is for example used for the benchmark application in chapter 6.

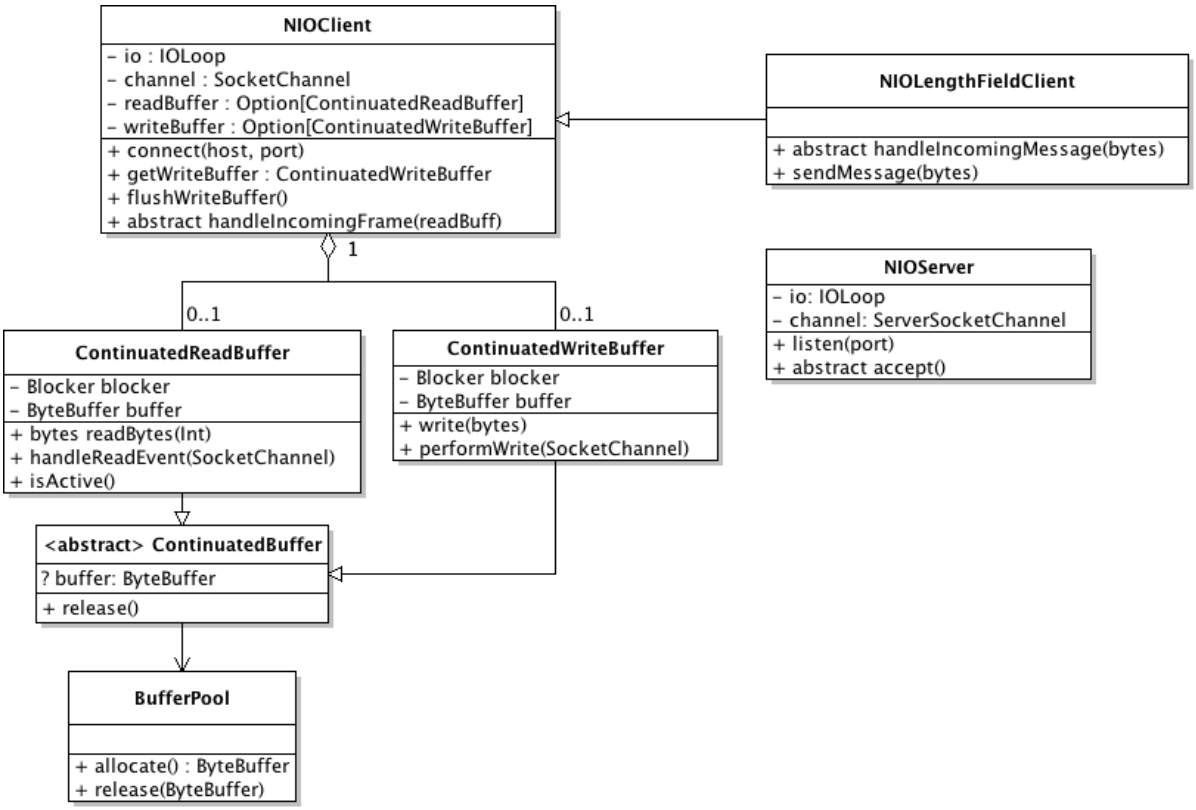


Figure 4.2.: Class diagram of the client and server classes including the buffer infrastructure.

4.2.4. Future objects

A `Future` object can either be in three states: Completed, Error or Pending. Because the future object itself is only a public entry point to the underlaying `Promise` object, the error or value is maintained inside this `Promise`. The `Promise` class is also responsible for creating the corresponding future objects. A future itself can create new promises for combined events.

It is decided to implement the `transform` and `onComplete` methods. The basic idea of both events are the same, but the `transform` methods always returns a new future with the result of the callback function, while the `onComplete` only accepts a `PartialFunction` that does return a `Unit`. The use of the `PartialFunction` class ensures that the user of the framework is able to perform *case matching* inside the passed function body.

Next to these methods, Scala requires the `filter`, `map` and `flatMap` in order to be able to create `for`-comprehensions. The result of the provided functionality is exactly the same as future comprehension in the Akka project as shown in Listing 3.5.

It is important to handle failures that happen inside the callback methods, this because the callback methods should never throw exceptions into the framework itself. This is performed by the `Wrapper` singleton object inside the `future` package. This object provides a `catcher` method that catches all exceptions and transforms these into normal return values. Such a returned exception can be stored inside the *promise* that owns the *future*.

Now, this future library is exactly the same as traditional ones since it supports comprehension and attaching of handler functions. Listing 4.1 shows how easy this functionality can be extended by the more developer friendly `get` method. The only limitation caused by the Continuations library is that the method cannot throw exceptions, and the system was thus forced to return an `Either` object instead.

Listing 4.1: The use of continuations inside the future *get*-method.

```
1  def get : Either[V, Throwable] @suspendable = shift {
2    ret => this._promise.registerExecution(ret)
3  }
```

4.2.5. Thread-safe access

The event loop framework itself is designed for single threaded usage only. To be able to work in an environment with multiple external threads, as described in subsection 2.2.1, it is required to have a thread-safe wrapper around the single-threaded interface. This ensures that the single threaded environment is never bothered by any performance overhead caused by mutual exclusions.

The singleton object `mainLoopTS` takes this responsibility of wrapping the normal `mainLoop` object. It offers a similar interface, and will forward all calls to the `mainLoop` object in a thread-safe manner. The `mainLoop` object offers the `intercept` method to register an execution from the `mainLoop` thread in a thread-safe way. The implementa-

tion of `intercept` is shown in Listing 4.2 and the `mainLoop` only tries to execute the `interceptFunction` property every event loop iteration. The nice aspect about this is that the iteration itself does not has to acquire any lock, but just reads the variable and tries to execute the intercept function. Normally, this could result into a race condition, but the `wakeup` signal used by `IOloop` ensures that the IO loop always wakes up the next iteration, even if it is already awake. In addition, reference assignments are atomically inside the JVM, so it is impossible that the `interceptFunction` variable is not completely written.

The `intercept` method requires obtaining a lock of two different monitors. The reason for this is that the `intercept` method itself is only allowed to be used by one single thread at the same time, and the monitor on `interceptFunction` is required to transfer the notification when the intercept call has been executed. In case the monitor on `mainLoop` itself only was used, the `wait` call at line 17 would release the lock which could cause another potential interceptor to overwrite the `interceptFunction` variable.

Listing 4.2: The *intercept* method of the *mainLoop* singleton object.

```
1   /**
2    * Thread safe method to register an intercept call at the mainLoop.
3    * The registered function will be executed by the mainLoop thread,
4    * and should thus not block.
5    */
6   def intercept(func: () => Unit) {
7     // Outer monitor is required because only one thread can register
8     // an intercept at a time
9     this.synchronized {
10      this.interceptFunction = Some(() => {
11        func() // Execute function, and notify whenever it is finished
12        this.interceptFunction.synchronized({
13          this.interceptFunction.notify()
14        })
15      })
16      this.ioLoop.wakeup() // IOloop might be blocking, so wakeup
17      this.interceptFunction.synchronized {
18        // Wait until the intercept signals the completion
19        this.interceptFunction.wait()
20      }
21      this.interceptFunction = None
22    }
23  }
```

4.2.6. Project dependencies

Apache Maven is used for managing the dependencies of all created Java and Scala projects during the thesis. The *EventLoop* itself does not have any real dependencies, only *ScalaTest* is used to create some test cases of the `IOLoop`. Finally, the *Eclipse* IDE is used for development, but this not strictly required.

4.3. Results

The comparison in chapter 6 will discuss the details of this project with respect to code quality and system performance. However, it can now already be concluded that the continuations are a useful construct for both low level suspending of IO operations and high level blocking on responses. However, two important issues occurred during the implementation that impact the developer friendliness in a very negative way. The next two subsections will discuss these problems that resulted into the inspiration for the project in chapter 5.

4.3.1. Coupling of continuations

In the ideal way, all existing software components should be able to use this event loop architecture without any modifications. This is of course quite difficult since the communication of the proactor pattern is completely different than the structure of normal blocking communication. This because the proactor pattern requires at least either future objects and/or callbacks to make asynchronous communication possible, as already explained in section 3.2.

However, with the use of Continuations the abstract code structure is again the same (or at least similar) to the blocking way of programming. This raises the important question: *Why is it not possible to make the event loop architecture compatible with the standard blocking APIs available in the Java/Scala programming language?* In this way, it would be possible to use for example an old legacy system that reads and writes messages from a stream without any change. Please note that these libraries do not things in parallel (because the libraries only knows about blocking communication and not about Continuations), but it would still make the system more scalable with respect to the total number of connection compared to the normal blocking way (i.e. there is no thread per connection and dead connections do not maintain any allocated buffers).

Java (and thus Scala) provide several low level APIs for communication. The most commonly accepted classes for socket communication seems to be `OutputStream` and `InputStream`. For example, the Google Protocol Buffers API offers two functions for protocol buffer messages as shown in Listing 4.3.

Listing 4.3: Java input and output interface of the Google Protocol Buffer project [46].

```
1 // serializes the message and writes it to an OutputStream:
2 void writeTo(OutputStream output);
3 // reads and parses a message from an InputStream:
4 static Person parseFrom(InputStream input);
```

It would be very useful if our scalable event loop architecture could just respectively pass the `ContinuatedWriteBuffer` and `ContinuatedReadBuffer` to these functions. For the Input- and Output stream it is respectively required that the abstract methods `int read()` and `void write(int b)` are implemented. Currently, the continuated read and write buffers offer respectively the methods `read(numberOfBytes: Int): Byte` and

`write(bytes: Array[Byte])`. So, this maps quite well to each other, and now it should only be a manner of implementing the standard stream classes in our framework.

Unfortunately, this is not the case. One thing that the above story did not tell is that suspended methods do have a special return type. Scala continuations require that a return type of a suspended method is wrapped by a `@CPS`-annotation. Concretely, this means that the above return types are appended with a `@CPS[.., ..]`-annotation, and thus does not map to the abstract input and output stream methods of the standard Java API.

The only possibility to completely encapsulate the continuations is to add a `reset` inside the overridden method because this removes the suspendable `@CPS` annotation. However, this is not possible in the `read()` method because a return value is required. So, this means that it is impossible to encapsulate the continuations inside a new implementation of the existing blocking stream classes.

4.3.2. Continuations and Exceptions

As shown in Listing 4.1, the suspended method is required to return an `Either` object that contains either an exception or the actual value. This is caused by the limitation that exceptions cannot be thrown from a `shift` structure in order to be caught in the normal way in the calling context. This issue is known by the Scala community, and some less elegant solutions are already invented [47].

The biggest problem of the returned `Either` object is that it is not possible to just dereference the future value inside, for example, an `if` statement. This was already not always possible due to the `@CPS[.., ..]`-annotation, which seems to be not completely compatible with all basic language constructs. But now the system actually forces to always extract the value by a `match` transformation before the actual computation.

Another issue is caused by the underlaying callback structure of the continuations, which makes it still impossible to have linear stack traces. This is a problem that can probably not be solved easily, since it is an important concept of the continuations.

5. A hybrid solution

In chapter 4 is shown that it is possible to write asynchronous code in a pure linear way that flows from top to bottom with the use of co-routines. However, subsection 4.3.1 explained that Scala Continuations are not able to become fully compatible with the blocking API's. This chapter will give a solution that will be fully compatible and is still able to parallelize outgoing requests. Furthermore, the provided solution is expected to still adopt the most benefits from asynchronous event loop architectures. The only expectation is that there will be some minor performance overhead in some cases. The results of this solution and the comparison with other solutions can be found in chapter 6.

5.1. Analysis

The initial idea for this solution emerged from the previous mentioned Scala project. As explained in the previous chapter, Scala offers the `reset` and `shift` commands from the Continuations API. These language constructs allow the developer to write code from top to bottom, while the API ensures that an application will continue after the end of the `reset` scope in case a `suspended` call cannot be directly completed. The system will continue where the `suspended` call halted as soon as the `shift` operation can be completed (i.e. whenever the callback to the `reset` call is executed).

At first hand, this way of programming seems to be revolutionary. Your programming language will 'temporary store the local programming state', and later on 'continue with this context when the developer decides to return'. However, in fact this is just a developer friendly implementation to allow non-preemptive scheduling by allowing the developer to manually control the programming stack of the application.

The following subsections will show the analysis of a more low level approach, that makes applications fully compatible with existing blocking input/output stream API's.

5.1.1. Non-preemptive scheduling

Non-preemptive scheduling can be achieved by only re-scheduling whenever a thread (or programming context) cannot continue, or when a task is completely finished. In all these cases the active context decides itself when it releases the CPU. The major benefit of this approach compared to preemptive scheduling is that the entire system does not have to use any locks or mutual exclusions because the developer knows that the CPU belongs to him. In this case, context switches only happen during read or write operations.

Technically spoken, it is possible to distinguish two type of functions: reentrant and thread-safe functions [17]. From the application developer point of view, it is in case of non-preemptive scheduling still required to write thread-safe code, however, the code does not have to be reentrant anymore[1]. Reentrant functions require that a function can be executed simultaneously by multiple threads. Thus, this requires that access to all global or shared variables must be carefully designed. In contrast, a thread-safe function does not acquire that two functions can write to the same piece of shared memory at the same time. The usual solution for making a function reentrant is to require a lock (or monitor) before the function can be accessed. This is not required anymore in case of non-preemptive scheduling, which saves computation overhead of expensive locking and a lot of concurrency complexity during the development.

Listing 5.1 gives a very simple example of a C++ function that is not reentrant and thus requires a lock to prevent race-conditions in case of preemptive scheduling. Of course, this simple example could easily be solved by using a thread-safe Set library. But again, this requires the acquirement of a lock every time an `add` operation is executed. Please, be aware that if the code would be executed with non-preemptive scheduling, as in the proposed solution, it is already correct in the way it is given in Listing 5.1.

Listing 5.1: Simple function that is not reentrant.

```
1  // Shared variable between handlers
2  std::vector<int> shared;
3
4  void handler()
5  {
6    // Remote function that could require a context switch
7    int value = remoteRequest();
8    // The following code is execute atomically
9    if (std::find(shared.begin(), shared.end(), value) == shared.end())
10     // Value is not in shared
11     shared.push_back(value);
12 }
```

The example in Listing 5.2 shows why a developer must still write thread safe code in case of non-preemptive scheduling. The provided piece of code is not reentrant, but this time also not thread-safe code. In this case, another call to `handler()` could potentially change the `shared vector` during the context switch of the second request at line 11. Now, the requirement that `value` is not present inside `shared` could not hold anymore, and a race condition is born.

Listing 5.2: Simple function that is not reentrant, but also not thread-safe.

```
1  // Shared variable between handlers
2  std::vector<int> shared;
3
4  void handler()
5  {
```

[1]Please note that the previous chapters did not make the distinction between thread-safe and reentrant. These chapters used term thread-safe in both cases.

```
  // Remote function that could require a context switch
  int check = remoteRequest();
  // The following code is execute atomically
  if (std::find(shared.begin(), shared.end(), check) == shared.end())
  {
    doAnotherRequest(); // requires IO, so might cause a context-switch
    shared.push_back(value);
  }
}
```

5.1.2. Context switching

Cooperative context switching can be achieved by using a continuations library as explained in subsection 4.1.2. However, as mentioned, a lot of libraries do not allow to completely encapsulate the controls of context switching. In order words, the the application developer will always be bothered by the fact that there is a difference between blocking and 'continuated' code. To overcome this problem, it is required to take a low level approach that actually allows to manually control the program stack. At first hand, this sounds very complex and thus very risky to implement as a small product of a master thesis. But lets first sum up which operations are required for a 'context' to implement a simple continuation API that could solve our scheduling problem:

constructor It should be possible to create a new programming stack including all additional information about the program counter etc. Also, it should be possible to specify an operation that should execute whenever a running computation of a context is finished. Finally, the entire application should not terminate when a context finishes, in this case another suspended context will be started instead.

suspend() The active context should be stored inside the context object. After this call another context should start with another task, the new task (or some potential new child of this task) is responsible to call the `continue()` method at the suspended context at some point.

continue() Finally, it should be possible to resume to a stored programming state of an earlier suspended context. When this operation will be performed, the active running context should be destroyed. The difficulty of this step is that a context should never be destroyed without doing some additional cleanups. At the point the active context will call the `continue()` method of a suspended context it might have allocated some dynamic memory on the heap. This kind of memory should of course be released after the context is destroyed to, prevent memory leaks. Additionally, it is important to observe that even garbage collected languages might in some cases not be aware about the destroyed stack depending on the type of garbage collector and the way of context switching. Thus, also in a garbage collected environment it is still very important to take this cleanup procedure into account.

From the historical viewpoint, a lot of similar issues happened with the use of non-local jumps [59]. In the past, these control flow manipulators were used for basic exception handling to jump up in the stack.

The most reasonable solution to overcome all these issues is to just let an active context completely terminate before the system will resume to the suspended context.

The scheduler should support the following operations:

currentContext Get the active context that can be used as a reference to perform the `switchBackTo` operation.

switchBackTo(suspendedContext) Will mark the current context as finished, so it can be terminated and be destroyed after it has been finished. After terminating the current context, it will call the `continue()` method of the context so that one will become active.

suspend() Will be called as soon a task cannot be completed and thus should be stored temporary. Next to suspending the active context, it will initialize and start a new context.

Please note that the above description does not provides any implementation details, but is just intended to give a high level idea of the required functionality of the context switching part.

The operations above already provide a less complex view on this at first hand over-complex looking approach. Except for the potential memory leak issues, the functionality is quite trivial. This saying under the assumption that there exist a well designed API that is able to satisfy the above requirements.

5.1.3. Blocking IO streams

Most programming languages provide blocking IO classes in their standard API's. For instance, Java comes standard with the `java.io` package [24], and C++ offers the `IOStream` library [3] by default. Additionally, Java provides the `java.nio` (Java's new I/O) [25] package for non-blocking and selector support. As already explained in subsection 3.1.2, the new I/O package is incompatible with old blocking API, while the goal of this approach is to make the blocking classes completely compatible with non-blocking IO.

In order to make these two worlds compatible with each other, the integration with the standard API's is an important proof. How the integration with the blocking stream API's is performed depends on the design of the of the API. For Java and Scala the best solution seems to be to derive the `InputStream` and `OutputStream` classes. These are both abstract because the Java I/O stream classes heavily depend on inheritance for making specializations, as shown in Figure 5.1.

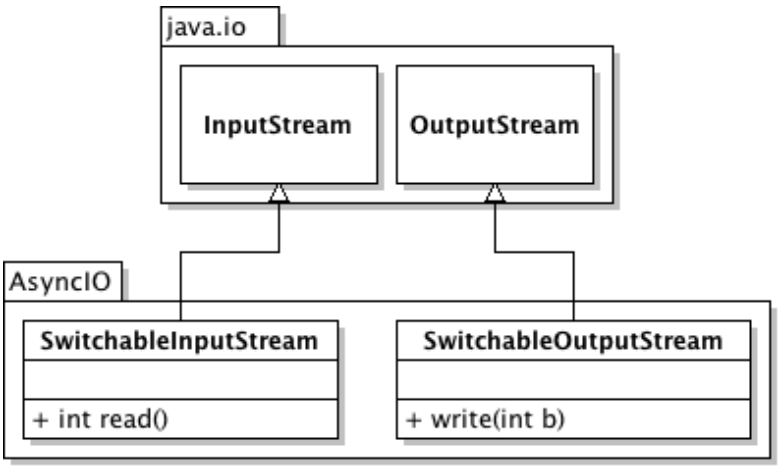


Figure 5.1.: Class diagram of the inheritance of the Java IO stream classes. The AsyncIO package represents the future stream implementation.

In contrast, the C++ `IOStream` library is not designed to create any new `IOStream` specializations for supporting new kinds of devices [62, p. 339]. The C++ `streambuf` class together with the `IStream` and `OStream` classes implement the *chain of command* software pattern. The rational behind this decision is to have an additional layer between the devices and the actual IO-logic inside an application. This concretely means that only two new implementations of the `streambuf` classes are required for our goal, as shown in Figure 5.2.

In both cases, the implemented stream classes are responsible for maintaining the usual read and write buffers, but also for controlling the scheduler. This means that a stream object also maintains a reference to the scheduler, and thus to the event loop that owns the stream. The goal is that the event loop is completely encapsulated for the classes that perform IO operations. The details of the implementation will be given in section 5.2.

5.1.4. Parallelization of outgoing requests

An important goal is that outgoing requests over different connections can be parallelized. An outgoing request commonly exists of two parts: a write operation of the request, followed by a read operation of the response.

Write operations In principle, write operations do never *have to* block because it is in theory possible to directly push the data of a write operation to the write queue. This way, write operations are automatically 'parallelized'. However, in some cases it is also desirable to block for a write buffer flush, for instance, to validate if the connection is still valid after the write operation is performed. Another reason could be to flush a full buffer which prevents allocating a new one.

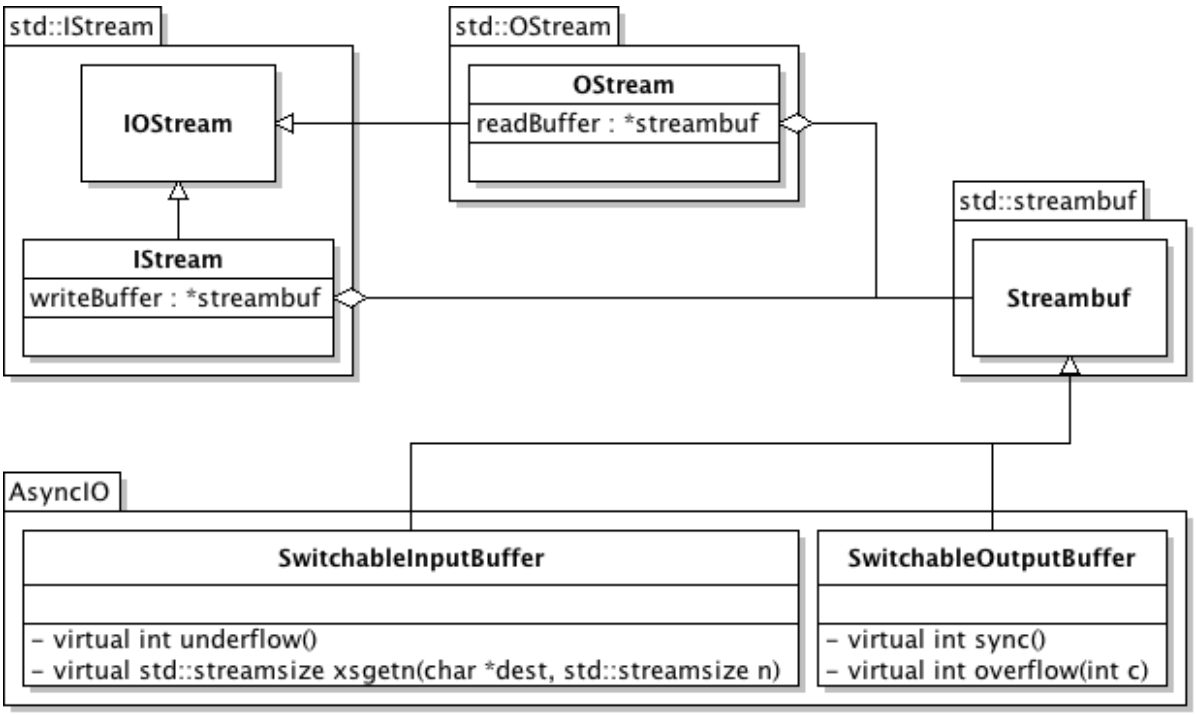


Figure 5.2.: Class diagram of chain of comment pattern of the C++ IOStream library. The AsyncIO package represents the future buffer implementation.

Read operations It is impossible to parallelize reads operation over different connections in the same programming context (i.e. on the same programing stack) without introducing some additional asynchronous programming logic. This because a read operation always requires that the read value is written to some sort of local variable, which is impossible to parallelize in a by the developer controlled manner. In other words, the developer should have the controls to indicate that a read operation should start, followed by a control to say that the future value must be ready to use.

The typical implementation that solves the above problem is a future placeholder that is already treated in previous chapters. The additional requirement of this placeholder is that it is not mandatory at first hand to support attaching of *onReady* / *onComplete* events. Instead, it should support a blocking retrieve method that will suspend until the actual value is ready.

5.1.5. Existing projects with cooperative IOStream support

The following two existing projects also aim on cooperative scheduling of an event selector by providing normal `IOStream` support. Though, both projects do have slightly different objectives compared to this project.

The GNU Portable Threads [16] As already mentioned, this is a very mature project. The major liability of this project is that it uses the less scalable `select` selector

instead of, for example, `epoll`. In addition, it does not provide any high level design to really improve the programming complexity.

Mozzy Mordor [37] Same as for the GNU Pth project, the project does not really aim on high level distributed programming. Another important difference with the intention of this project is that scheduling of a new thread should only happen when really necessary. In contrast, the Mordor project just creates a new context for every incoming connection, and does not allow to control the underlaying asynchronous behavior anymore.

5.2. Implementation

The suggested solution is implemented in the C++ programming language. The most important reason for this decision is that this approach requires low level controls for stack switching. The C programming language already offered this functionality by the POSIX `ucontext.h` library, which can seamlessly be wrapped by a C++ class. The details of this are given in subsection 5.2.3. Most other programming languages that have API's for creating simple non-preemptive threads (e.g. fibers or tasklets) do not result into a simpler implementation compared to the `ucontext.h` library. This because the required controls for scheduling are not very complex as already explained in subsection 5.1.2.

5.2.1. Eventloop

This eventloop implementation follows the architecture of the Scala event loop implementation in the previous chapter, and is also split up into an execution queue and separate IO-loop. A sequence diagram of this structure is shown in Figure 5.3. In addition to the Scala implementation, the *MainLoop* class also consists of a context scheduler, as shown in Figure 5.4. Because the event handlers, that are derived from `IOlistener`, are able to do blocking operations, it is possible that a context switch happens during the execution of an event.

In other words, `onRead()`, `onWrite()`, `onClose()` or the execution functions passed to the *ExecutionQueue* are able to do a stack switch at any moment in time. This means, that the entire call-path to these functions should be written in a thread-safe way. This means that it should not store shared data on the stack or modify a 'shared' variable after a potential context switch could happen. In practice, this means that both the `ExecutionQueue` and `IOLoop` classes must be carefully designed with respect to thread-safety. A concrete example of this is shown in Listing 5.3, in here the pop operation could normally be performed after the execution. However, in this case the call to `execution` could cause a context switch. This would cause that the pop operation is not executed before a switchback happens. This would probably result in an endless loop because the same `execution` function will be executed over and over.

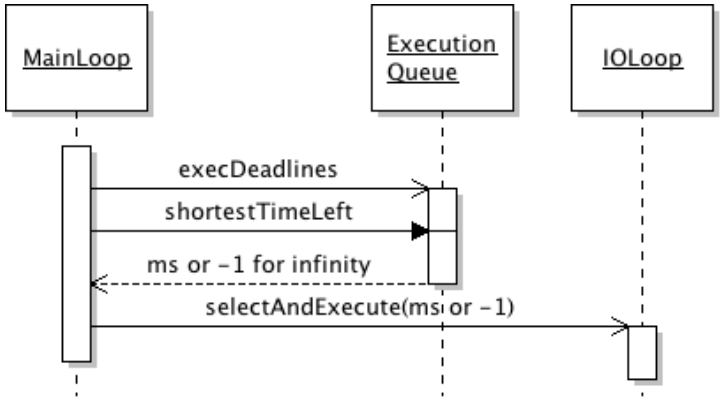


Figure 5.3.: Sequence diagram of one iteration of the MainLoop.

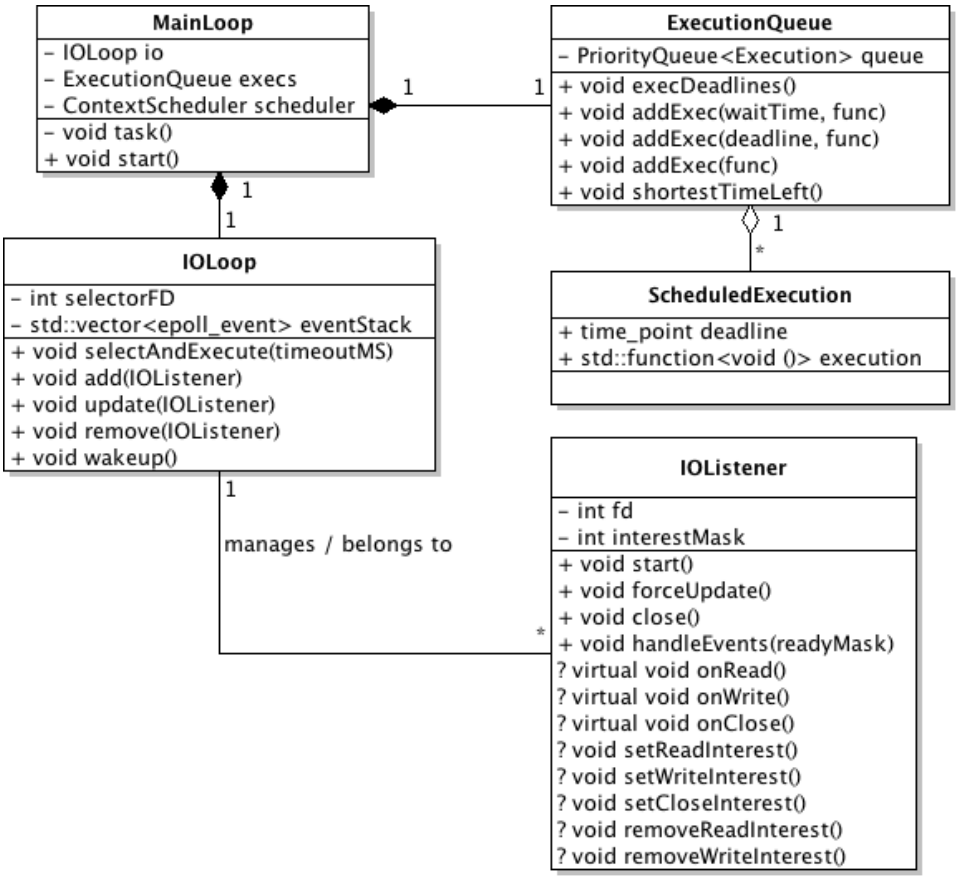


Figure 5.4.: Class diagram of most important event loop classes. Some functionality of the *MainLoop* class is left out for simplicity reasons.

Listing 5.3: *execDeadlines()* function inside the *ExecutionQueue* class. The pop operation at line 6 must happen before a context switch happen in order to not execute the same function twice.

```
1   void execDeadlines()
2   {
3     while (!d_queue.empty() && d_queue.top().deadline <= std::chrono::
        system_clock::now())
4     {
5       // top.execution() might be suspended, so first pop it (so it will
          not be executed twice!)
6       auto top = d_queue.top();
7       d_queue.pop();
8       top.execution();
9     }
10  };
```

Another similar example is shown in Listing 5.4, this function is actually split up into two parts. First the operating system will be queried for ready events, and after this the handlers of the ready events will be executed. From the code perspective, the `select` function call at line 3 will query the operating system as long as there are events ready, and put these inside the shared `d_eventStack`. After this phase is finished, the handlers of the events inside the stack will be executed one by one. Again, important here is that the pop operation must be performed before a potential context switch might happen.

Next to thread safety there is another reason for doing the `select` and `execute` phases into two separate steps. The `select` call will query a fixed amount of events from the operating system by the `epoll_pwait` system call, in this implementation this amount is specified by the constant static value `s_maxEvents`. Because this is a fixed amount, it is possible that the maximum number of events is reached while there are more events ready. In this case, the operating system will just pass the maximum number of ready events to the application. The problem here is that it is possible that always the same events will popup, while other events starve [8]. By copying the events, it ensured that the selector is always empty after the procedure, thus this potential starvation risk is prevented. The liability of this approach is of course that the event data is always at least copied twice.

Listing 5.4: *selectAndExecute()* function inside the *IOLoop* class. The *select* function will put all retrieved events inside a shared stack to ensure that if a context switch happens the next worker will just continue with the first not treated event. Additionally, copying the event to a separate queue also prevents starvation in case the operating system repeatedly passes the maximum number of events.

```
1  void IOLoop::selectAndExecute(int timeoutMS)
2  {
3    if (select(timeoutMS)) // Select as many events as possible to prevent
       starvation
4      while (select(0)); // Do not wait again a second time
5
6    while (!d_eventStack.empty()) // handle all events that are selected
```

```
7    {
8      auto event = d_eventStack.back();
9      d_eventStack.pop_back();
10     execute(event);
11   }
12 }
```

5.2.2. Open connections

Connections that will be managed by the `IOLoop` class can either be opened by the application itself (outgoing connections) or by a server socket (incoming connections). In this case, the server socket is managed by the same `IOLoop` instance as well. The listener sockets and connection classes are all derived from the `IOListener` class, this base class allows to control the interests of the active connection, as seen in Figure 5.4.

Because the goal of the system is to support a high number of open connections, it is an important aspect that an IDLE connection itself does not consume a lot of memory. This is implemented in such a way that a connection will not have any allocated read or write buffers, until it is actually receiving or sending data. Also, the `onNewIncoming` method of the `AsyncClient` class allows to register an event handler for new incoming messages. This method prevents that the programming stack must be preserved between multiple subsequent messages of an incoming client.

5.2.3. Context scheduling

The class diagram of all context switching related classes is shown in Figure 5.5, one `ContextScheduler` object is owned by the `MainLoop` class which is displayed in Figure 5.4. The `Context` and `ExecutorContext` classes are both wrappers around the earlier mentioned `ucontext.h` library. The specialized class `ExecutorContext` maintains, in addition to `Context`, a separate program stack. This in order to be able to execute a function on this stack, and specify the return address of another stack that will be schedules as soon the function terminates. These actions are performed by a call to `makecontext`.

One risky aspect of this last call is that it is not specified by the manual if it can be executed multiple times. In the current implementation, it is required that the resume address of the stack (i.e. the `ucontext_t *uc_link` field) can be set dynamically at any point of time. This is caused by the fact that the resume address is only known at the moment the scheduler schedules a termination. Because it was required that a context always completely terminates to prevent memory leaks as already mentioned in subsection 5.1.2. Currently, this does not seem to give any problems in the test environment, but on other platforms it could result into dangerous bugs.

Context Scheduler The most important controls of `ContextSchulder` class are the `suspend` and `scheduleSwitchBack` methods. Which is quite similar to offered functionality of the high level `reset` and `shift` functions provided by the Scala Continuations

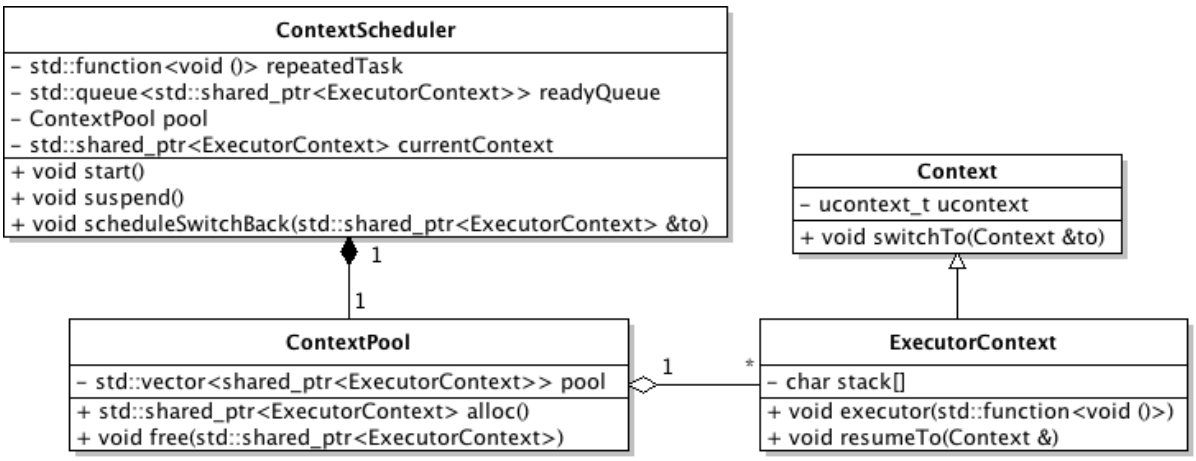


Figure 5.5.: Class diagram of all context switching related classes.

API. The only difference is that the caller of the `scheduleSwitchBack` method is responsible for passing a reference to the right context object, which is in Scala just a simple high level callback.

When the `ContextScheduler` class is created, it will receive a reference to a function by its constructor. This function, in our case the event loop task defined in `MainLoop`, is executed repeatedly as long as the scheduler is not stopped. An architecture that is often used to schedule such tasks is the *worker pool* pattern. For example, threaded event loop systems often use a thread pool with a fixed number of worker threads that will 'simultaneously' pop requests from a shared task queue. For this implementation a different architecture is chosen, primary because non-preemptive scheduling is used.

The implemented architecture is displayed in Figure 5.6, the actual code that implements this loop is given in Listing 5.5. This approach ensures that always the minimum number of context objects are allocated, and thus the minimum number of memory is consumed. Though, it needs to be remarked, that the implementation allows to keep a number of terminated contexts. The reason for this is that memory recycling improves performance.

In addition, this architecture ensures that the system does not have to perform a stack switch for every new task-iteration. Instead, it respectively only suspends or switches back, when the current context really cannot continue or another process is able to continue.

Listing 5.5: The *taskWrapper* function that manages the executions of the repeated task and the termination of the current context.

```
1 void ContextScheduler::taskWrapper()
2 {
3   while (d_readyQueue.empty())
4     d_repeatedTask();
5   auto resumeTo = d_readyQueue.front();
6   d_readyQueue.pop();
7   d_currentContext->resumeTo(*resumeTo);
```

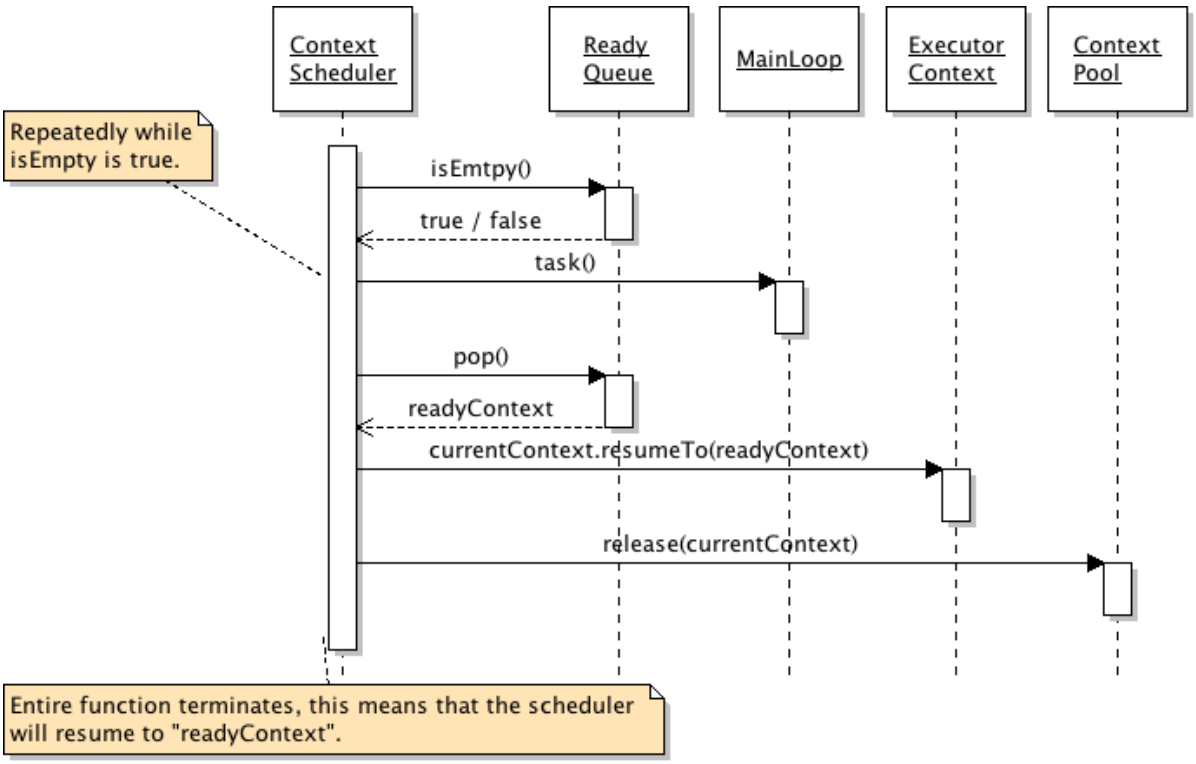


Figure 5.6.: Sequence diagram of the entire lifetime of a context that is managed by the *ContextScheduler*.

```
8    d_pool.release(d_currentContext);
9    d_currentContext = resumeTo;
10 }
```

Context states The internal implementation of the scheduler is quite similar to, for example, a normal operating system process scheduler. The possible state changes are shown in Figure 5.7. From the implementation point of view, the ready state results into a ready queue that is managed by the `ContextScheduler`, i.e. as soon as a context is passed to the `scheduleSwitchBack` function it is placed into this queue. Before a repeated task is executed, it will always validate if this queue is empty. If this is not the case, it will terminate the current context, and release itself to the `ContextPool`. As shown in Figure 5.7, terminated contexts can be used again, this task is full filled by the `ContextPool` class.

The `ContextPool` class maintains a configurable maximum number of terminated contexts. The benefit of this is that the operating system does not have to allocate new stack space for every new context. The current implementation does not clean a terminated stack (i.e. make all values zero). But for security reasons this could be required in real world situations.

An important implementation detail is that a context releases *itself* to the `ContextPool` before it terminates. This means that the `ContextPool` should never immediately free such an object, but do it during the next iteration. The current implementation solves this by always first creating space into the pool, before the latest released object is placed in there. In this way, if the pool is already full, an old object will be released, instead of the latest one.

Finally, the system must maintain the suspended / blocked states as well. This is done in an implicit way by the maintainer of the callback that will call `scheduleSwitchBack` because the caller is responsible for registering the switch back event. This is done by copying the reference to the current context (e.g. by capturing the variable by value in a lambda function), and passing it to `scheduleSwitchBack` at the moment the event fires. Because the reference is managed by the reference counted `std::shared_ptr` wrapper [52], the object will not be freed in the meanwhile.

5.2.4. IOStream and StreamBuffer classes

Subsection 5.1.3 explained that the `IOStream` library provides the standard blocking stream classes in the C++ language. The previous mentioned `AsyncClient` class already provides the `blockingRead` and `blockingWrite` methods that will respectively suspend the call until the data is actually read or written. Now, these functions only need to be wrapped in order to make the system completely compatible with the `IOStream` package.

Wrapping is performed by implementing a specialization of the `streambuf` class as already shown in Figure 5.2. Now, the context switching is completely encapsulated for the rest of the system, which usually uses the `OStream` and `IStream` classes. In fact, from the stream classes point of view, the implementation is exactly the same as with normal

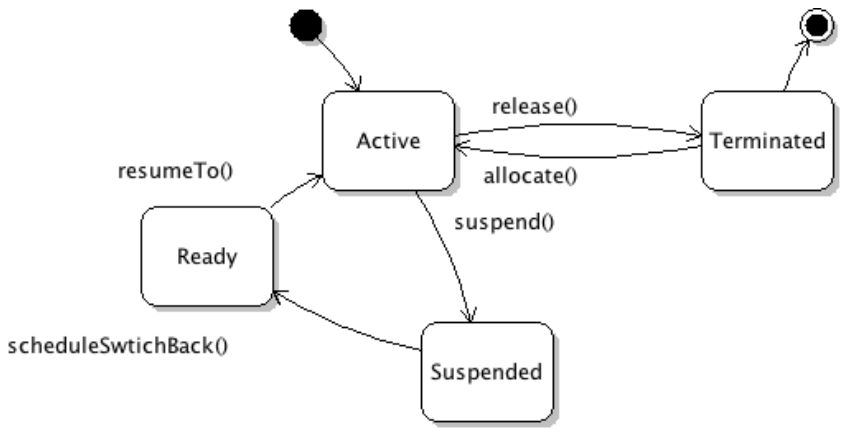


Figure 5.7.: State diagram of all possible states of a context during a lifetime. Only one context can be active at the same time.

blocking operations, except that the context switching is not handled by the operating system but by the application itself. This way, the benefits of asynchronous communication are still there because the operating system still manages open connections with an efficient selector. Also, the `streambuf` objects are only allocated if a connection is writing or reading, and released as soon they become empty. Finally, the use of the `IOStream` classes also directly resulted into a lot of 'free' functionality. A nice example of this is 'tying of streams'. The standard stream classes allow to tie an output stream to it, this connected stream is always flushed before an operation is performed on the owner of it. This gives the important benefit that a developer does not have to worry about manual stream flushes when performing multiple alternating reads and writes.

Another interesting point is, that while the system is completely compatible with the blocking stream classes, the developer is *not* forced to always use this functionality. For example, to implement a protocol that has a fixed length header, it is not enforced to allocate a `streambuf` class for reading the initial header of each message frame. In this case, it is more efficient to first register a read operation of the fixed length header. Whereafter it is still possible to create the `streambuf` object at the moment the data is arriving. This hybrid approach is possible because the functionality of `AsyncClient` client can still be used next to the `istream` buffer. Of course, in this case it is important to validate that the `streambuf` is empty as soon as all data is parsed. If this is not the case, then the next header might already be loaded into the `streambuf` object, and should thus be used instead. This behavior is, for instance, implemented by a simple while loop inside the `onNewIncoming` method.

5.2.5. Future's

Now, it is proven by the `IOStream` approach of the previous chapter that the system is completely compatible with all blocking API's. However, the other requirement was that outgoing requests could be parallelized, which is not possible with normal extraction op-

erators. It is possible to use the low level `read` and `write` functionality of `AsyncClient`, but this would highly complicate development for the developer because it has to work with a buffer and a low level asynchronous callback based API. This would result into all kinds of synchronization problems.

For this reason, an approach with future placeholders that work with the normal extraction operators is recommended. This is implemented by the `Future` template class, that has the value that should be extract as a template type argument. For instance, Listing 5.6 shows three examples of types of future objects that can be constructed. A complete example of a non-blocking extraction with future objects is shown in Listing 5.7.

Listing 5.6: Some non-blocking extraction examples with different kind of future values.

```
Future<int> intExtraction; // Extracts a single int

// Extracts a single word (depending on active stream settings)
Future<std::string> stringExtraction;

// Note: the OtherObject class supports extract operatations
// Extracts a single OtherObject instance
Future<OtherObject> objectExtraction;
```

Listing 5.7: A complete example of a non-blocking extraction of two words from two different streams. Please note that it is in this case completely useless to use future instead of a normal value extraction. However, this functionality might become useful as soon as the future is for example returned by a function that writes a request before it reads the results. Now this function can return the future directly without blocking, while the client is allowed to just do another request in the meanwhile.

```
1 Future<std::string> aWord, anotherWord;
2
3 stream1 >> aWord;
4 stream2 >> anotherWord;
5
6 cout << "The result of stream1: " << *aWord << '\n'
7      << "The result of stream2: " << *anotherWord << '\n';
```

Just a placeholder The future object itself is nothing more than a placeholder of the futuristic extracted value. As soon as the extract operator of the future object is called (line 3 and 4 in Listing 5.7), the only thing it performs is a registration of the futuristic extraction at the stream class. After this registration it just continues without extracting anything. The future object itself maintains a reference to the stream where it should extract the value from and a `shared_ptr` to the dynamically allocated object. This pointer is `null` while the promised value is not yet present. The `shared_ptr` ensures that the dynamically allocated object is freed as soon as no futures are pointing to the value anymore. This was required because future objects support copying by the copy constructor.

Maintaining order of extractions The `AsyncReadBuffer` class is responsible for maintaining the order of the future extractions, since multiple non-blocking extraction might happen consecutively. This ordering is ensured by a queue that maintains all future extract operations, in the form of normal function references (e.g. C++11 lambda functions). A unique pointer value, that is provided by the future object, ensures that the future extraction are executed until the right place because it is possible that the future objects are not dereferenced in order.

In theory, it is also possible that a blocking extraction is performed right after a non-blocking read is performed. The framework will always perform a flush of the non-blocking read queue for this reason.

An important point to realize is that the extract operator of the `Future` class only works on `IStream` instances that have a `AsyncReadBuffer`. This is not validated at compile time in the current implementation, instead a `dynamic_cast` is used to validate at runtime if the actual given buffer is an instantiation of class `AsyncReadBuffer`.

Pointers to the future Finally, the futuristic value that is maintained by the `Future` object is required at some point in the computation. At this point, the analogy with a normal pointer is chosen. A normal pointer is dereferenced with the dereference operator (`*`) or arrow operator (`->`), as soon as the value is required instead of the memory address. The same holds in the future object implementation, as soon as the value is required, the dereference or arrow operator can be used to retrieve the value as shown in line 6 and 7 in Listing 5.7. If the value is not yet present, the system will execute the blocking operation that was registered, as explained in the last paragraph.

Higher level future functionality It some cases it might be useful to still have the transformation possibilities as the Scala project offered. Especially because attaching events is expected to be cheaper than a complete context switch which performed by the dereference and arrow operators.

Because this behavior requires a completely different future architecture, it is not yet implemented for this project. Now, the future objects only store a futuristic 'blocking' execution, while combining future objects would also require to execute the normal callbacks instead.

5.2.6. Thread safe access

The entire event loop system is designed to be single threaded. This also means that the framework is designed without any lock or mutex, and uses libraries that are not thread-safe. Though, as mentioned, the project should still provide thread-safe access to the outside world. For this functionality, exactly the same intercept principle could be used as explained in subsection 4.2.5, but this is currently not implemented for this project. On the other hand, the wakeup procedure to interrupt a potential blocking `epoll_wait` call is implemented.

An old fashioned way to create such a wakeup mechanism is a so called 'self-pipe' trick [43]. In this case, the application opens a connection to itself that will wakeup the blocking `epoll_wait` call. Because this approach was quite cumbersome to implement and a potential performance loss in case of heavy use, operating system manufactures introduced new blocking select calls that support signal interruption.

The current implementation uses the `epoll_pwait` system call for this [9]. As soon as an `IOLoop` instance is constructed, it will block all interrupt signals. This ensures that interrupt signals are only delivered during the `epoll-pwait` call. The custom `SIGUSR1` signal is used for this, and should thus not be used for any other purposes. Now, another application is able to interrupt the blocking event loop by sending the `SIGUSR1` to the event loop application. In addition, the application can also execute the `kill` system call [30] with its own `pid`. Which is actually performed by the `wakeUp` method of the `IOLoop` class.

5.2.7. Debugging

The goal of coroutines is to have linear code *and* linear execution to make debugging easier as well. However, because there are multiple stacks in our application, it is not obvious that all debugging tools will work seamlessly together with it. The primary debugging tool used during this project is the Valgrind Project [56]. Valgrind offers additionally functionality to monitor multiple dynamically allocated stacks, as shown in the example where Valgrind cooperates with the `ucontext` library [11].

In order to let our framework cooperate with with Valgrind as well, the `d_valgrindContext` property is introduced in the `ExecutorContext` class. This property, and the Valgrind header dependency, should of course be optional if the project would be used in real environments. This is not supported in the current version.

5.2.8. Project dependencies

This section lists the project dependencies. For this project platform independence did not play any role because it was just an experimental setup. This is also the reason why the system is only designed for and tested at Linux versions 2.6.19 or higher. Currently, all programming code is compliant to the new C++11 standard. Aspects of this new version that are heavily used are lambda functions, `nullptr`, `shared_ptr`, and the `auto` keyword. The following table sums up all the dependencies of the project. In addition, it also provides an exit strategy for each dependency in order to remove it from the project.

Dependency	**Required for**	**Exit strategy**

Linux 2.6.19	Some specific calls to socket systems calls like `read` and `write`. In addition also `sys/epoll.h` and `ucontext.h` which are described below.	See Linux-only dependencies below.
`sys/epoll.h`	The `epoll_pwait`, `epoll_create` and `epoll_ctl` system calls used by the `IOLoop` class.	This part of the system could be easily be replaced by an existing event loop system. For example, *libevent* [33] and *libev* [32] could implement this part of the system. The reason to not use these API's already is because it is not 100% clear if these libraries will not give any issues with the context switching part of the project. In addition, these libraries might already be thread-safe, which could influence benchmarking in a non-fair way.
`ucontext.h`	Context switching functionality that is performed by the `Context` and `ExecutorContext` classes.	A normal non-preemptive scheduling thread library could be used to replace this functionality. A popular example of such a project is the GNU Pth Thread library [16]. Such a replacement could also solve the incompatibilities with operating systems that do not support or deprecate the `ucontext.h` library.
`valgrind /valgrind.h`	To monitor all different stacks created by the `ucontext.h` library the Valgrind project requires additional support at programming level. The `ContextExecutor` class is responsible for manually registering and unregistering programming stacks.	Valgrind integration is completely optional, and a compiler directive could easily ensure that it is only used in case the developer requires this functionality.

`signal.h`	Interrupt single of blocking `epoll_pwait` call in `IOLoop` class.	A comparable signal mechanism, or the already mentioned 'self-pipe' trick. Of course, in case of a multi-threaded architecture, it would also be possible to remove this 'wakeup' part of the system.

During development, the system is only tested on Debian 6.0.4 with Linux kernel 2.6.32-5-amd64. Compilation of the C++ code is performed by G++ 4.6 with EGLIBC 2.13-32. An attempt to also run the framework on Mac OS X 10.6.8 failed, the observed failure caused an endless loop which was probably caused by `ucontext` dependency. This dependency gives also a deprecated error during compilation.

Currently, the project is build with the SCons build tool version 2.0, which is similar to the well known `make` tool. The file `SConstruct` is located the source root of the project. This configuration file specifies all compiler and linker arguments. In order to make use of Valgrind support, the `-g` flag must be appended to the `cppFlags` variable.

5.3. Results

In contrast to the Scala project of the previous chapter, this project was a really experimental approach. The most important goal of the project is to be a working proof of the intended concept. The intended goals was to come up with a hybrid system that is able to cooperate with old blocking API's while still providing asynchronous controls.

Despite of the current implementation status, it can still be stated that it is proven that the `IOStream` library is able to encapsulate the event loop. Also, the extension with the future library that allows to extract value from a stream in a non-blocking way. Finally, in contrast to the expectation, the implementation complexity of the event loop and scheduler part was not even really difficult. Actually, the design of a developer friendly entry point in the system (e.g. simple clients and server, construction of streams) can be considered as more complex.

The most important liability of this approach is that a simple piece code already would require at least one context switch. However, two important side notes should be placed with respect to this. First of all, the destruction-flush optimization for write buffers ensure that there will be no context switch when the buffer is destroyed and still contained some data. Secondly, the expectation is that usually an entire message arrives in one iteration. In this case, a context switch is also not required because the system has already all data read in the buffer at once. This means that in many cases, if an incoming requests does depend on additional outgoing requests, just zero context switches happen.

6. Experiments and Evaluation

The last two chapters show two different implementations that provide the ability to write synchronous understandable code but still have the ability to control the asynchronous behavior of the application. Next to these asynchronous controls, both frameworks use an internal event selector for dispatching the IO events, which often means that it is not required that a thread or process halts until an IO operation can be performed. As explained in chapter 1, the expectation and goal is that this results into much more scalable applications with respect to the number of connections. Now, the arising questions are: *Does the use of coroutines impact the performance and scalability in a significant way compared to the other approaches explained in chapter 3?*, and *Does the use of coroutines really improve the programming complexity for the developer?*

6.1. Benchmark requirements

The questions above cannot be answered by a single simple benchmark. For both questions the same distinction should be made as in chapter 3. This because the assessment of a few low level IO operations on a single connection has completely different requirements than the assessment of a few high level remote requests. In the last case, the first assessment will still play an important role (i.e. all requests will still perform the low level IO operations by using coroutines) but now the programming context has to be preserved for a much longer time. And because the memory consumption is the expected bottleneck, this amount of time will surely impact the performance.

Another important aspect are the qualities of interest that will be measured during the benchmark execution. The most important qualities are shown in the following list:

Maximum number of connections The most important goal of the project is scalability with respect to the number of open connections. It should thus be measured what the maximum number of connections is in different situations. The expectation is that the maximum number of connections is highly related to the memory consumption of the application.

Maximum number of messages per second or another kind of throughput measurement is required to assess the performance of each connection. This quality is of course highly related to the number of open connections. However, this would still give an indication about if there is any significant overhead in case of switching between a lot of open connections that require a high throughput.

Response time might be an important measurement if the system would be used in real life situations. Especially under very heavy circumstances this might become an important requirement.

CPU consumption The actual CPU consumption is an important aspect to indicate the actual bottlenecks of the system. In addition, the information on what the system is working on could give a good indication when the system runs out of memory.

Memory consumption Same as for the CPU consumption. Moreover, memory consumption is the expected bottleneck for the coroutine approach.

Fault/Error rate Especially if the system would (almost) run out of resources, the errors measured by the client side is an important indicator of the robustness of the system. Furthermore, an important question would be how the (server-) system recovers after a period of abundant throughput.

The above qualities of interest are in the ideal case measured in real world situations. However, this can of course only be simulated in our case. Suggestions for such realistic implementation are listed in the following table:

Simple request-response The request-response pattern is a commonly used pattern in networking. This pattern can either be implemented by keeping the underlaying (tcp-) connection open between multiple requests or by closing the connection after each completed request. The prior is principle is, for example, used by the HTTP 1.0 protocol. Because this project aims on long open connections, this is the most important point to assess at first hand.

A simple benchmark could be a (shared) storage system which can be accessed by multiple clients at the same time.

Publish subscribe A powerful pattern that gains popularity on the web is the publish subscribe pattern. The difficulty of such a system is that the number of subscribers might be really high, while a publication should still arrive within a certain amount of time. The publish subscribe pattern can be as best implemented by keeping the connection open for a long time. A heartbeat or ping-echo mechanism must be used in case of reliable connected subscribers.

Typical business layer or crawler/collector client could really benefit from the use of coroutines with respect to the high level future functionality explained in chapter 3. The actual system performance would be an interesting aspect because a single programming context must be preserved for a longer time in this case.

Due to limited time, the actual possibilities for benchmarking were also limited. For this reason, it is chosen to first only implement the simple request-response pattern. This will not proof the feasibility of cooperative scheduling in high level distributed programming, but it will demonstrate if there is a significant performance or scalability impact for blocking IO operations compared to the pure event based approaches.

6.2. Setup

Because it is not feasible to benchmark the system in a real server infrastructure, it is chosen to perform the tests on a single machine. This gives on the one hand the benefit that the actual network infrastructure does not influence the results. But on the other hand, both running applications could influence the behavior of each other.

Also, because the system is tested on a normal laptop, other running application could influence the results as well. In order to minimize this, it is decided to run all tests inside a virtual machine. The CPU power of the guest operating system is capped by the virtual machine to ensures that small operations of the host operating system will not impact the benchmark. In the ideal case, the benchmarks would be executed multiple times, and the results of these separate runs would be combined to increase the accuracy. This is not performed because there were no real differences observed between different manual runs.

All tests are performed on Debian 6.0.4 with Linux kernel 2.6.32-5-amd64 inside VirtualBox installed on Mac OS X 10.6.8. The virtual machine uses one core with a 40% CPU cap and 256MB memory. This might at hand first seem an unrealistic configuration for a server system. Though, this configuration is similar to a typical small VPS setup, and is for example equal to more or less 50% of the resources provided by an Amazon micro instance. Another reason for providing less resources to the benchmark system is to make the system limits earlier visible. In other words, the maximum number of open connections and maximum message rate will highly increase when more resources applied, but the expectation is that the rest of the system behavior will not differ from a system with more resources.

In order to be able to handle a lot of open connections the standard Debian configuration must be slightly modified. A linux installation is by default not allowed to handle many open files and open sockets. This for stability and security reasons. The maximum number of file descriptors is limited both system-wide and at user level, respectively configured by the files `/etc/sysctl.conf` and `/etc/security/limits.conf`. In addition, the C++ and Scala projects are respectively compiled with the `-O3` and `-optimize` flags, for the highest optimizations with respect to both memory and CPU usage.

The tools `pidstat` and `sar` are used to measure and store resource statistics at least every 10 seconds. The benchmarks itself are performed by starting a client that will increase the number of open connections to the server in a linear way. Every 200 milliseconds a new connection is started that will send a request every 30 seconds. The server is expected to reply within 20 seconds, and only in that case the connection is defined as 'valid'.

Finally, it should be determined when the benchmark is finished and everything can be terminated. Because the expectation is that the system will run out of memory whenever a certain amount of connection is active, the point the virtual machine starts to swap would be a good indicator. However, the actual stop criterium is when the number of failed connections increases in a significant way.

6.3. Implementation

To ensure that the actual limits of the event loop framework become visible, it is important that the benchmark algorithm itself does not have a dominant impact on the system performance. A complex benchmark algorithm could for example also highlight other performance differences within the used programming languages, which is not the goal of this project. Though, future benchmarks could use more realistic implementations to indicate how much the event loop performance influences the to total system performance.

Client-server communication In this benchmark it is chosen that every connected client 'owns' a counter inside the server. The client will send an *increase request* every 30 seconds and validate if the received response matches with the expected value. The server will maintain the counters inside a hash map that maps every key (a normal character string) to a signed integers (of at least 32 bits). Before the value will be increased, the system should first check if the counter exists. If this is not the case, it will directly insert the received value since it assumes that an unset counter is zero.

Protocol The benchmark uses a normal tcp sockets for communication between the client and server. The JSON protocol is used for the exchanging messages, where each message is handled in FiFo order. A typical request and response are displayed in the following two lines:

```
{"field": "client-key", "value": 1}
{"currentValue": 1, "isNew": true}
```

Two of the four implementations described in the upcoming subsection really require that a four byte length header (big endian) is prepended in front of the actual payload. This is forced by the event loop structure as already explained in section 3.1.

6.3.1. The systems

Four different server implementations and two different client implementations are created to be assessed by the described benchmark algorithm. This section will describe the different implementations for both the server and client implementations. The initial idea was to connect the same client to each server application, and to measure if the there is any significant difference in performance. However, because there two different communication protocols, one with and one without length header, there must be two different clients.

Thread per connection server

This is the traditional approach for server systems, and very likely the implementation with the lowest performance [4]. It uses a preemptive scheduled thread per connection

and traditional IO operations. The use of preemptive scheduled threads requires a thread-safe implementation of the *counter* hash map because multiple threads could potentially perform an increase operation at the same time.

Because a blocking system does not know whenever new incoming data arrives it has to wait for data continuously. The basic principle the worker thread will repeatedly is; wait for incoming data, read and parse the data, create the response and finally wait until the response is written. This is as best demonstrated by the lines of code of the `CounterRequestHandler` in Listing 6.1. Both `writeOutgoing` and `readIncoming` classes are able to use respectively the blocking `DataOutputStream` and `DataInputStream` implementations, which highly simplifies the parser and generator code as for instance in Listing 6.2. This implementation and the upcoming Scala implementations make use of the Jerkson JSON library [27].

Finally it should be mentioned that in theory the blocking implementation did not require the fixed length header to parse message. This was added to be compatible with the Netty and Scala projects discussed in the next section.

Listing 6.1: The Scala code of the connection handler inside the worker thread.

```
1  private def worker() {
2    while (true)
3      writeOutgoing(handleIncoming(readIncoming()))
4  }
```

Listing 6.2: The very simple blocking implementation of *readIncoming()*.

```
1  private def readIncoming() = {
2    val size = in.readInt()
3    val payload = new Array[Byte](size)
4    in.readFully(payload)
5    parse[IncrementOperation](payload)
6  }
```

Netty server

The Netty project, which is already discussed in subsection 2.4.2, supports implementations for either fixed length protocols, length field based protocols, or delimited protocols. Both delimited as length field based protocols are in theory possible for our JSON messages. Also, it should be denoted that it is impossible to parse the raw JSON protocol messages without any of the mentioned 'tricks' in this project. Only an asynchronous implemented JSON protocol parser or a risky *trial-and-error* approach could still support the plain JSON structure, which are of course both not obvious choices.

Almost the entire Java implementation in Netty is provided in Listing 6.3 and Listing 6.4. A first observation is that the entire implementation is relatively small for a reactor based approach. This is highly caused by the protocol modification and the feature rich standard library of Netty.

Also in this project it is required that the hash map implementation is thread-safe because there are multiple request handler threads. The *JsonCounterRequestHandler* is

shared among all connections because it maintains the shared counter state.

Listing 6.3: The *JsonCounterPipeline* factory class.

```
1  import org.jboss.netty.channel.ChannelPipeline;
2  import org.jboss.netty.channel.ChannelPipelineFactory;
3  import org.jboss.netty.channel.Channels;
4  import org.jboss.netty.handler.codec.frame.*;
5  import org.jboss.netty.handler.codec.string.*;
6  import org.jboss.netty.util.CharsetUtil;
7
8  public class JsonCounterPipeline implements ChannelPipelineFactory {
9
10   private LengthFieldBasedFrameDecoder lengthDecoder = new
       LengthFieldBasedFrameDecoder(Integer.MAX_VALUE, 0, 4);
11     private StringDecoder stringDecoder = new StringDecoder(CharsetUtil.
         UTF_8);
12
13   private LengthFieldPrepender lengthEncoder = new LengthFieldPrepender(4)
       ;
14   private StringEncoder stringEncoder = new StringEncoder(CharsetUtil.
       UTF_8);
15
16   private JsonCounterRequestHandler reqHandler = new
       JsonCounterRequestHandler();
17
18   public ChannelPipeline getPipeline() throws Exception {
19     ChannelPipeline p = Channels.pipeline();
20     p.addLast("length-decoder", lengthDecoder);
21     p.addLast("string-decoder", stringDecoder);
22         p.addLast("length-encoder", lengthEncoder);
23         p.addLast("string-encoder", stringEncoder);
24         p.addLast("json-encoder", reqHandler);
25     return p;
26   }
27 }
```

Listing 6.4: The *JsonCounterRequestHandler* implementation.

```
1  import java.util.*;
2
3  import org.codehaus.jackson.map.*;
4  import org.jboss.netty.channel.*;
5
6  public class JsonCounterRequestHandler extends SimpleChannelHandler {
7
8    // Ensure counter is thread-safe since it is accessed by multiple
       handler threads at the same time
9    private Map<String, Integer> counters = Collections.synchronizedMap(new
       HashMap<String, Integer>());
10   private ObjectMapper mapper = new ObjectMapper();
11
12   public static class IncrementRequest {
13     public int value;
```

```
14      public String field;
15    }
16
17    public static class IncrementResult {
18      public int currentValue;
19      public boolean isNew;
20    }
21
22    private IncrementResult handleIncoming(IncrementRequest request) {
23      IncrementResult result = new IncrementResult();
24      if (this.counters.containsKey(request.field)) {
25        result.currentValue = request.value + this.counters.get(request.
            field);
26        result.isNew = false;
27      } else {
28        result.currentValue = request.value;
29        result.isNew = true;
30      }
31      this.counters.put(request.field, result.currentValue);
32      return result;
33    }
34
35    public void messageReceived(ChannelHandlerContext ctx, MessageEvent e)
        throws Exception {
36      // Get string and remove header four bytes
37      String msg = ((String) e.getMessage()).substring(4);
38      ObjectReader reader = mapper.reader(IncrementRequest.class);
39      IncrementRequest incoming = reader.readValue(msg);
40      IncrementResult response = this.handleIncoming(incoming);
41      String responseMsg = mapper.writeValueAsString(response);
42      e.getChannel().write(responseMsg);
43    }
44  }
```

Scala event loop server

As explained in subsection 4.3.1, the continuations are not compatible with 'normal' blocking calls. This forces the benchmark protocol to also prepend the message with a sequence of length header bytes. This is implemented in the same way as with Netty, and both protocol implementations are compatible with each other.

The server code is shown inside Listing 6.5, but the actual benefits of the continuations are encapsulated inside the derived `NIOLengthFieldClient` class that is displayed in Listing 6.8.

Listing 6.5: The *JSONCounterServer* implementation.

```
1  import eventloop._
2  import java.nio.channels._
3  import com.codahale.jerkson.Json._
4  import scala.collection.mutable._
5
```

```
6  case class IncrementOperation(field: String, value: Int)
7  case class IncrementResult(isNew: Boolean, currentValue: Int)
8
9  class JSONCounterServer(channel : SocketChannel, server:
       JSONCounterListener
10     ) extends NIOLengthFieldClient(mainLoop.ioLoop, channel) {
11
12   def handleIncomingMessage(payload: Array[Byte]) {
13     val msg = parse[IncrementOperation](payload)
14     val existing = server.counters.get(msg.field)
15     val newValue = existing.getOrElse(0) + msg.value
16     server.counters.put(msg.field, newValue)
17     val returnMsg = IncrementResult(existing.isEmpty, newValue)
18     this.sendMessage(generate(returnMsg).getBytes())
19   }
20
21 }
22
23 class JSONCounterListener extends NIOServer(mainLoop.ioLoop) {
24   val counters = new HashMap[String, Int]
25
26   def accept(channel: SocketChannel) {
27     new JSONCounterServer(channel, this).startReading()
28   }
29 }
```

Listing 6.6: The *NIOLengthFieldClient* implementation.

```
1  package eventloop
2
3  import java.nio.channels.SocketChannel
4  import scala.util.continuations._
5
6  /**
7   * Helper class for Clients that will work with fixed messages
8   * instead of streams.
9   *
10  * The class first reads four big endian bytes followed by the
11  * actual payload of that length. The handleIncomingMessage
12  * should be implemented by the client.
13  */
14 abstract class NIOLengthFieldClient(ioLoop: IOLoop,
15     channel : SocketChannel = SocketChannel.open()) extends NIOClient(
           ioLoop, channel) {
16
17   final def handleIncomingFrame(buff: ContinuatedReadBuffer) {
18     reset {
19       val msgLength = buff.readFixedInt()
20       val payload = buff.read(msgLength)
21       this.handleIncomingMessage(payload)
22     }
23   }
24
```

```
  protected def handleIncomingMessage(payload: Array[Byte])

  def sendMessage(payload: Array[Byte]) {
    reset {
      val buff = this.getWriteBuffer()
      buff.writeFixedInt(payload.length)
      buff.write(payload)
      this.flushWriteBuffer
    }
  }
}
```

C++ stream compatible server

In contrast to the previous projects, the C++ project provides the real human readable approach with respect to the JSON protocol. This because no length has to be prepended to the actual payload. JsonBox [29], the used JSON library, provides the standard insertion and extraction operators in order to be fully compatible with the `iostream` classes.

The main function of the project is shown in Listing 6.7. The most important goal that this code proofs is that the event loop mechanism actually cooperates with the standard stream classes.

Listing 6.7: The *main* method of the C++ implementation.

```
int main()
{
  AsyncServer server(mainLoop, [](shared_ptr<AsyncClient> newCon) {
    connections.insert(newCon);
    newCon->start();
    newCon->onNewIncoming([](istream &in, ostream &out) {
      JsonBox::Value incoming(in); // Parse incoming json obj
      auto incomingObj = incoming.getObject();
      if (incomingObj["field"].isString() && incomingObj["value"].
          isInteger())
      {
        // Valid message, thus handline incoming message.
        auto incResult = counter.increase(incomingObj["field"].getString()
            ,
          incomingObj["value"].getInt());
        JsonBox::Object outgoing; // construct outgoing message
        outgoing["currentValue"] = incResult.currentValue;
        outgoing["isNew"] = incResult.isNew;
        JsonBox::Value(outgoing).writeToStream(out, false);
      }
      else // Invalid incoming message, return error response
        JsonBox::Value("Invalid Request").writeToStream(out, false);
      // Remove possible whitespace characters from the stream prevent
      // direct blocking of the next iteration.
      //WhiteSpace::removeOptionalWhitespace(in); // COMMENTED OUT FOR
          WHITESPACE EXPERIMENT
```

```
    }, true);
  });

  mainLoop.executions.addExec([&] {
    int port = 8080;
    cout << "Start server at port " << port << endl;
    server.start(port);
  });
  mainLoop.start();
}
```

Scala event loop client

The event loop client implementation in Scala is responsible for starting the new connections that have to send the periodic messages. It uses the `addTimedExec` method provided by the `mainLoop` singleton for these timing aspects. In addition, the created future functionality provides a `withDeadline` method that constructs a new future placeholder. This future object will automatically invalidate after a configurable amount of time in case the value is not (yet) received.

Listing 6.8: The *increaseAndCheck()* method inside the *CounterConnection* class.

```
  private def increaseAndCheck() {
    val fut = this.client.increase(IncrementOperation(token, 1))
    fut withDeadline(timeoutMS) onComplete {
      case Left(IncrementResult(_, v)) => {
        this.currentValue += 1
        if (v == this.currentValue)
          reschedule()
        else
          ClientMain.deadClients += 1
      }
      case Right(e) => ClientMain.deadClients += 1
    }
  }
```

C++ stream compatible client

The client implementation of the C++ project is surely not as extensive as the Scala approach in the previous section. For example, it does neither support future objects with deadlines nor exceptions. But this could be extended if more time would be spend to this project. Though, the goal to have cooperative scheduling in combination with providing compatibility with blocking IO libraries is fully proven by the current implementation.

6.4. Performance

This section will discuss the performance results of all server implementations above, the first three are executed with Scala client, and the last one by the C++ client. The

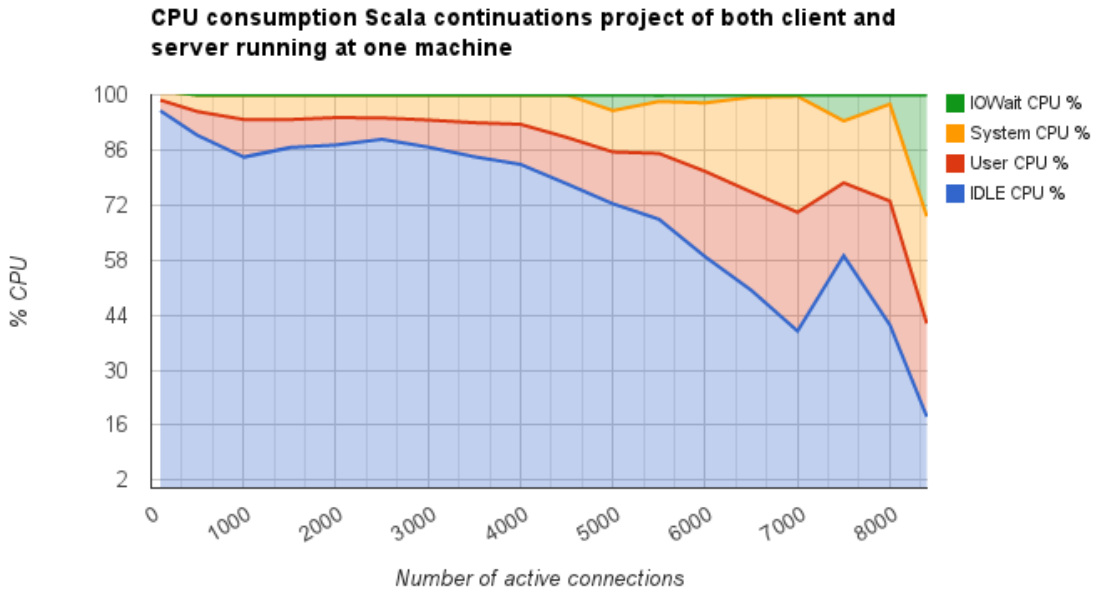


Figure 6.1.: Trashing will influence the system performance after more or less 7000 active connections in the Scala server and client benchmark. The first significant *IOwait* time takes place after around 5000 connections.

formatted measurements can be found in Appendix A.

As expected, at least three of the four runs did finally suffer by *trashing*. The virtual machine indicated this by having many disk IO operations, and an increased *IOWait* wait time is observed in the CPU statistics. This typical behavior is graphically shown for the Scala client and server-run in Figure 6.1. This diagram shows a clear spike in *IOwait* time after 5000 active connections. This corresponds quite well with the observed errors which exploded at 8400 connections. The delay can be caused by the fact that an error usually gets detected after the 20 seconds deadline, or that the error statistics are only recorded after every 100 new connections. Finally, the spike in *IOwait* time itself might not seem so high, but it should be mentioned that this value indicates the percentage of total computation time wasted on wait time for IO operations.

6.4.1. Server side memory consumption

The diagram in Figure 6.2 shows the non-swapped physical memory consumption of the server processes. The first observation is that the thread per connection implementation is the less scalable of all. This because the amount of memory used for a single connection is extremely high. The thread per connection-run actually did not give any errors, but the client application crashed due to a segmentation fault caused by memory shortage.

Both Netty and our Scala implementation have similar performance with respect to memory consumption. The decrease in non-swapped memory after 5500 connection is probably caused by swapping and a garbage collector that tries to free as much memory

as possible. Noteworthy is that both Netty and Scala have a relative high amount of memory consumption in the very beginning. In addition, the memory consumption of the Netty project seems to increase faster than the Scala project. This is remarkable because the Netty project is actually fully event based, while the Scala project has to allocate space for the use of continuations. The C++ server implementation shows the most linear growth as possible, and has the lowest consumption in the beginning as well. Finally, it should be mentioned that the C++ test did not show any errors in the log file. However, both the server and client application did not respond anymore after 125000 active connections, and also no active logging was observed anymore.

Thus, if we finally look at the difference in memory growth between the Netty and C++ server implementations, we actually observe the opposite of the expected behavior. The increase in memory consumption between 500 and 5000 active connections is for Netty and C++ respectively $64752 - 32768 = 31984kb$ and $9948 - 2468 = 7480kb$. Approximately said, the memory consumption of the Netty project increases four times faster than the C++ project. Although, it should be mentioned that it is possible that the Netty project receives more physical memory due to the JVM. This might also explain the decrease in memory consumption after 5000 connections while the server is still operating properly.

The above numbers also mean that a single client consumes approximately $7.1kb$ and $1.6kb$ of physical memory for respectively the Netty and C++ implementation. This is far less than the 8912 bytes ($8kb$) that one single context consumes[1]. This is caused by the very short life time of a context when a message arrives. And in many cases the context does not even have to be switched because of the optimizations listed in section 5.3.

6.4.2. Client side memory consumption

The memory consumption of the two clients is shown in the diagram in Figure 6.3. Again, the C++ implementation shows the most linear growth in memory consumption, and a much lower start consumption. Remarkable in this case is that the memory consumption of the client is quite similar to the memory consumption the server, while the client has to maintain the allocated coroutine contexts for a longer period per connection. This because it will send the request and then wait for the response in the same programming stack. This would give the expectation that the client would require signification more memory in total.

[1]The actual stack size might differ per operating system because it is defined by the constant *SIGSTKSZ*.

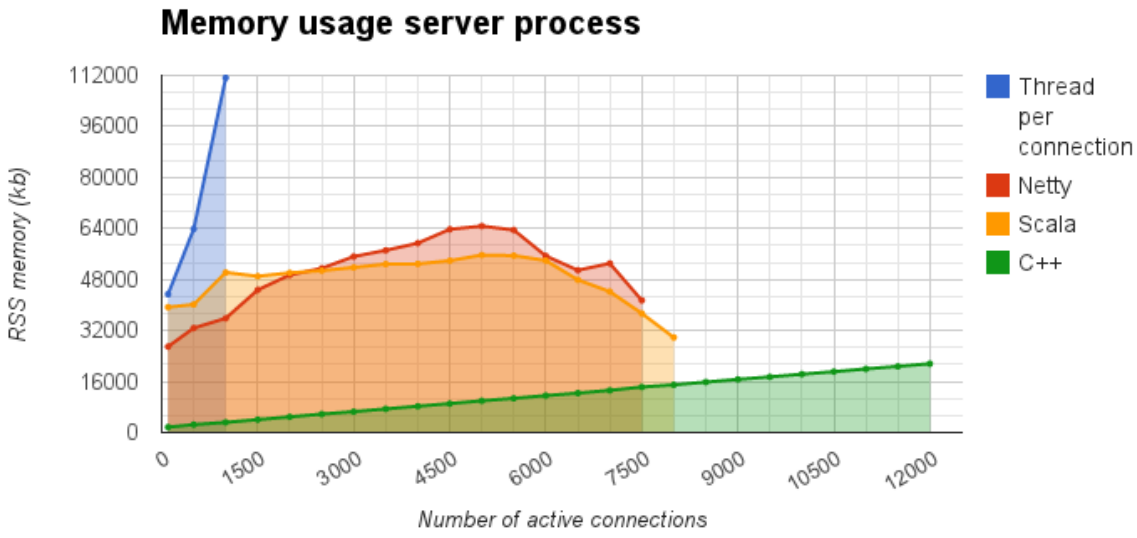


Figure 6.2.: Non-swapped physical memory consumption of the server processes.

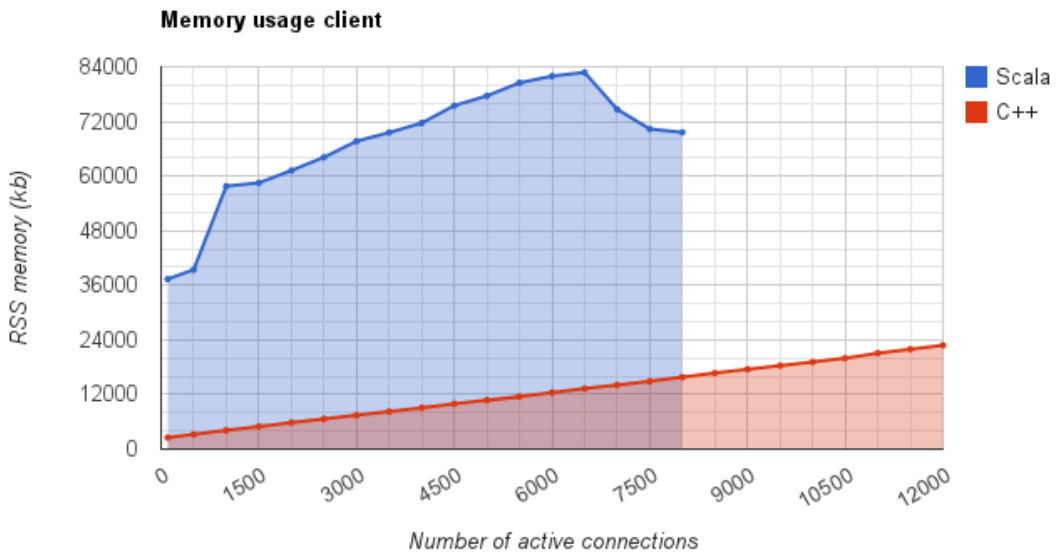


Figure 6.3.: Non-swapped physical memory consumption of the client processes.

7. Conclusion

This chapter evaluates the entire project. This is done by first providing a retrospective of the entire course of the project. Thereafter a discussion is given with the conclusions of the project. Finally, the chapter ends with a future work discussion.

7.1. Retrospective

In the very beginning, the research started already with the focus on developer friendliness. Although, at this stage the focus was more on the benefits that can be taken from a strictly defined (service-) interface compared to dynamic 'typed' ones. After discussing the first results about datatypes that can be exchanged between services, it became more an more clear the traditional callback approach did not satisfy the developer friendliness requirement. This conclusion drastically changed the research direction.

Thereafter, the research started to focus more on the questions stated in the introduction. This was supported by a blogpost [65] and movie [67] that are proposing a similar approach. This resulted into the project treated in chapter 4. Some informal tests indicated that a *continuation*-switch is faster than a complete *stack*-switch. This was also expected because it is likely that the overhead of a continuation switch is almost comparable to a normal function call in Scala. This because every function is accessible as a normal heap object.

From this point of view this looked like a very promising approach. However, as explained in subsection 4.3.1, the suspended methods provided by the Continuations library can only be used in a transparent way for the developer. This gave the problem that it was impossible to implement the standard stream libraries in Java/Scala. Though, it should still be pointed out that continuations are very useful for the more high level structures. In this case, it actually provides the normal synchronous way of thinking for the developer. For instance, wrapping a normal callback infrastructure of a future library was very easy as seen in Listing 4.1 on page 58.

However, the linear code approach elicit that it from the low level data stream point of view can be very useful to have 'normal stream access'. As mentioned in chapter 6, it is even possible to implement protocols that are not possible in the event based structure without nasty tricks or slightly modifying the protocol. This finally resulted in the more experimental approach shown in chapter 5.

The streaming support approach in C++ was expected to be much more risky with respect to the implementation. This because it uses very low level context switching API's that was responsible for the continuous task processing structure. A complex side issue that occurred was that all event loop procedures which control the external

developer code had to be written in a thread-safe way. This because the event loop methods may never assume that it directly gains control about information on the stack after a potential context switch might happen. Though, probably because this problem was considered before the implementation started, only several small issues occurred.

Finally, the comparison of both projects and one or more existing projects became point of subject in order to answer the research questions. Due to limited time it was not possible anymore to do a real extensive benchmark with respect to code quality and performance. This would be an interesting aspect, especially to see when context switching of the C++ project might become a bottleneck with respect to performance. However, the aspects measured in chapter 6 show some interesting results.

7.2. Discussion and Conclusions

The following two sections will try to answer the research questions stated in the introduction of this document.

How can the complexity of asynchronous programming be improved without affecting the scalability in a significant way? The answer of this first research question is partly answered by the programming analysis. This chapter makes a distinction between *IO programing and design* and the more high level *Asynchronous application programming*. For both aspects it explains that the responsibility of the developer for maintaining the application state manually is the biggest issue of the developer. This is caused by the fact that it is impossible for the developer to think top-down way when programming, due to the forced state machine architecture.

For low level IO programming, this also results into the problem that using normal parsers becomes difficult. This because there are usually no asynchronous implementations available due to the increased complexity. The more developer friendly *proactor* pattern makes this a little bit easier. However, also in this case it still means that developers have to modify the intended protocol, or should rely on 'trial and error'-parsing. And even this nasty solution requires that the partial message of a failed parse attempt must be stored for another attempt manually. These difficulties are all in contrast with the ease of use of the standard stream API's we are used to use. We can conclude from this point of view, that *there is no easy and modular approach of low level non-blocking application programming without falling back to traditional streams.*

With respect to the more higher level of programming, the problem of asynchronous programming itself is exactly the same, but now another issue arrises as well. The example in the introduction in section 1.1 shows already that combining multiple events becomes an important aspect. This shows that in case of a change request the affected code range is much larger when the programming state has to be maintained manually.

In section 3.2 basically three different ways of high level programming are distinguished: *Blocking* (including synchronous code by coroutines), *callbacks* and *future objects*. In here is shown that composable futures, future comprehension and blocking code with asynchronous controls decreases the programming complexity in some cases.

Though, attaching events to future objects for handling responses always results again in a callback structure. This in contrast to blocking code with asynchronous controls (i.e. future objects) which just allows to dereferencing the value (e.g. as it were a normal pointer). This especially improves the complexity when a request depends on the results of multiple other requests because many condition-statements would again require complex nested code structures, even in case of future objects.

This way, it can be very useful to have 'normal' blocking code with the ability to parallelize outgoing requests. Now, the second question is if such an implementation would affect the scalability in a significant way. The benchmarks in section 6.4 showed that memory consumption is the bottleneck of all tested configurations. Moreover, this is also expected to be the bottleneck in many other cases as long as the system should be scalable with respect to the number of open connections, while the actually computations are not really CPU intensive. The benchmark also showed that the hybrid system did not consume a significant higher amount of memory for each connection, actually the opposite was observed. Although, as already mentioned, the system is not benchmarked in situations where the programming context must be stored for a longer time.

Is it possible to combine the benefits of an event based architecture with the ease of development that traditional stack-based programming provides? The hybrid context switching approach in chapter 5 actually proofs that this is perfectly possible. The implementation provides asynchronous controls in terms of future objects, that could theoretically be used to attach event handlers for simple response handling. Also, a future object provides functionality to block the current programming context until the expected response is received. In this case, the system will just continue with a new programming stack in the beginning of the event loop, and switch back as soon the expected event returns.

A major benefit of this approach is that a designer of, for example, a server framework is able to benefit from all asynchronous capabilities. For instance, the designer can just use the `onNewIncomming` event to handle new incoming messages of a server socket, but in this case the event handler can just use the `istream` and `ostream` objects as it were a blocking environment. Now, the system does not have to consume any stack memory for an IDLE connection as in a normal threaded system.

Concluding, it can be stated that this hybrid approach seems to be very promising from the programming perspective and the measured performance. Though, the following section will list some aspects that were not treated in this research, but should surely get attention to really proof all opportunities and pitfalls of this approach.

7.3. Future work

The following list summarizes the most important features, improvements and other aspects that were not treated in this research:

- The system is not tested with real DNS resolving. Both the Scala and C++ project are only used with `localhost` as hostname. This could be an important aspect

because a performance issue or bug in asynchronous DNS resolving could have an important impact on the system behavior.

- The C++ project does not support any high level future functionality.
  - Neither combining nor transforming events is supported. The current implementation really focussed on the compatibility with the IOStream library, while it would actually be mandatory to have high level future objects as well to measure impact on performance.
  - There are no timeout capabilities for creating deadlines for responses. The Scala implementation does this really elegant by providing the `withDeadline` transformer. A similar structure would be useful in this project.
  - Handling of exceptions in combination with future objects and event loop did not receive much attention yet.
- Support for parallelization of message processing on one connection. This can speedup the overall response time when some incoming requests have to wait for responses of outgoing connections. In addition, it would in this case possible to remove the FiFo requirement of incoming requests as, for example, MessagePack does [35].
- In some cases, a developer might forget about thread-safety, while this is really required if shared variables would be accessed within multiple (outgoing) requests. A well designed framework should ensure that developers are aware of this problem (e.g. now a developer does not see if a function might block or not), or that this problem is completely removed by the framework design (i.e. by not allowing to access variables within a continuated context).
- Finally, an improved benchmark setup that requires that a stack is preserved during two (or more) outgoing queries would consolidate the conclusion in the previous section.

A. Benchmark results

The following tables provide the summarized benchmark results. The information in these tables are used for the charts in chapter 6.

A.1. Scala client to 'thread per connection' server

Time	Created	Failed	IDLE%	User%	System%	IO%	S-CPU%	S-kb	C-CPU%	C-kb
15:23:44	100	0	94,28	3,92	1,81	0	3,21	43400	3,51	37484
15:25:06	500	0	87,18	7,37	5,45	0	5,79	63848	8,64	39460
15:26:50	1000	0	80,55	12,2	7,26	0	7,82	111260	11,21	55588
15:27:32	1200	0	73,38	10,02	8,81	7,79	8,51	119408	12,56	59700

A.2. Scala client to Netty server

Time	Created	Failed	IDLE%	User%	System%	IO%	S-CPU%	S-kb	C-CPU%	C-kb
13:59:27	100	0	91,59	5,67	2,74	0	6.98	26948	3.04	37932
14:00:49	500	0	91,52	4,75	3,74	0	4.94	32768	5.14	40000
14:02:32	1000	0	84,33	10,76	4,91	0	7.10	35816	9.05	59100
14:04:15	1500	0	88,6	7,67	3,73	0	6.87	44724	6.87	60052
14:05:58	2000	0	89,02	6,95	4,03	0	6.65	49416	7.66	62768
14:07:41	2500	0	87,21	8,06	4,73	0	6.96	51508	7.76	65644
14:09:24	3000	0	86,67	8,79	4,55	0	8.27	55208	8.68	68996
14:11:07	3500	0	84,75	9,62	5,63	0	8.66	57140	11.11	71016
14:12:51	4000	0	97,9	13,6	6,5	0	10.96	59396	13.10	73948
14:14:36	4500	0	70,77	18,27	6,68	4,28	13.26	63744	18.06	76900
14:16:21	5000	0	74,33	17,81	7,35	0,52	14.24	64752	17.46	78830
14:18:07	5500	0	65,47	21,37	11,7	1,46	16.84	63524	25.54	81076
14:19:55	6000	0	59,77	25,06	13,53	1,63	31.41	55380	40.6	80892
14:21:57	6500	0	60,17	23,84	15,55	0,44	38.94	50900	61.21	74080
14:24:34	7000	0	89,62	6,06	4,04	0	20.05	53048	24.5	67576
14:27:49	7500	0	89,79	5,86	3,94	0,4	26.17	41448	100.36	69336
14:28:51	7600	5824	82,96	8,15	6,86	2,04	4.44	38512	4.34	73152

A.3. Scala client to Scala server

Time	Created	Failed	IDLE%	User%	System%	IO%	S-CPU%	S-kb	C-CPU%	C-kb
14:38:14	100	0	95,98	2,71	2,31	0	2,72	39324	4,63	37360
14:39:36	500	0	89,75	6,03	4,22	0	4,52	40164	7,63	39420
14:41:19	1000	0	84,22	9,55	6,23	0	9,61	50200	11,49	57788
14:43:03	1500	0	86,64	7,09	6,28	0	5,45	49028	8,89	58508
14:44:46	2000	0	87,27	6,97	5,76	0	5,54	50076	10,08	61228
14:46:29	2500	0	88,71	5,44	5,84	0	4,98	50780	11,6	64136
14:48:11	3000	0	86,74	6,88	6,38	0	5,86	51784	10,1	67664
14:49:54	3500	0	84,27	8,67	7,06	0	6,33	52829	10,65	69564
14:51:38	4000	0	82,34	10,19	7,47	0	7,29	52892	12,66	71656
14:53:21	4500	0	77,4	11,81	10,8	0	9,3	53896	17,56	75480

14:55:05	5000	0	72,32	13,18	10,52	3,98	9,65	55680	18,58	77632
14:56:50	5500	0	68,37	16,75	13,22	1,6	12,09	55492	20,97	80492
14:58:35	6000	0	58,83	21,72	17,42	2,03	17,26	53940	31,31	81968
15:00:22	6500	0	50,29	25	24,13	0,58	24,54	47892	67,26	82784
15:02:18	7000	0	39,93	30,21	29,51	0,35	35,33	44196	76,34	74680
15:04:34	7500	0	59,12	18,51	15,75	6,63	40,23	37352	71,92	70348
15:07:14	8000	0	41,49	31,51	24,66	2,35	20,46	29820	40,51	69600
15:10:44	8400	5120	18,15	23,7	27,23	30,92	38,59	21664	49,85	70468

A.4. C++ client to C++ server

Time	Created	Failed	IDLE%	User%	System%	IO%	S-CPU%	S-kb	C-CPU%	C-kb
14:57:53	100	0	97,6	0,4	2	0	0,6	1696	1,8	1800
14:59:16	500	0	95,79	1,9	2,3	0	1,6	2468	3,81	2532
15:00:58	1000	0	94,18	2,91	2,91	0	2,41	3188	5,72	3248
15:02:41	1500	0	94,28	2,81	2,91	0	3,11	4036	7,53	4140
15:04:24	2000	0	92,98	4,01	3,01	0	3,21	4916	8,33	4952
15:06:06	2500	0	95,49	2,71	1,8	0	3,51	5768	9,02	5828
15:07:49	3000	0	92,26	3,92	3,82	0	4,3	6544	10,7	6640
15:09:31	3500	0	91,25	5,03	3,72	0	4,92	7404	11,66	7460
15:11:14	4000	0	90,25	5,13	4,62	0	5,13	8244	12,66	8256
15:12:56	4500	0	89	6,3	4,7	0	5,92	9080	13,64	9072
15:14:38	5000	0	87,7	6,7	5,6	0	6,2	9948	14,2	9964
15:16:21	5500	0	86,87	7,52	5,61	0	6,41	10728	15,13	10768
15:18:03	6000	0	87,06	7,02	5,92	0	6,32	11576	14,44	11556
15:19:46	6500	0	88,25	6,33	5,42	0	6,74	12356	15,59	12424
15:21:28	7000	0	86,69	8,11	5,21	0	7,04	13228	16	13316
15:23:11	7500	0	86,61	7,35	6,04	0	7,22	14272	16,85	14084
15:24:53	8000	0	84,15	8,73	7,12	0	8,22	14964	17,25	14916
15:26:36	8500	0	84,31	8,55	7,14	0	8,02	15816	18,25	15820
15:28:19	9000	0	81,9	10,31	7,79	0	8,55	16672	19,42	16728
15:30:01	9500	0	82,9	9,96	7,14	0	8,82	17448	18,84	17540
15:31:44	10000	0	81,28	10,11	8,61	0	9,34	18296	19,38	18336
15:33:27	10500	0	79,96	12,63	7,41	0	10,03	19084	21,16	19140
15:35:11	11000	0	76,35	13,46	10,19	0	11,04	19956	22,89	19992
15:36:55	11500	0	72,85	16,82	10,33	0	12,53	20752	24,04	21080
15:38:41	12000	0	66,6	20,82	12,58	0,6	16,58	21588	30,34	21956
15:40:32	12500	0	51,33	28,59	17,41	2,67	28,44	22240	50,2	22808
15:41:56	12800	0	49,25	29,19	21,56	0	37,2	22204	60,06	23920

Bibliography

[1] Avro c++ documentation. http://avro.apache.org/docs/1.4.0/api/cpp/html/index.html.

[2] Benchmarking libevent against libev. http://libev.schmorp.de/bench.html.

[3] C++ iostream library reference. http://www.cplusplus.com/reference/iostream/.

[4] The c10k problem. http://www.kegel.com/c10k.html.

[5] The chromium projects - inter-process communication (ipc). http://www.chromium.org/developers/design-documents/inter-process-communication.

[6] Comparing two high-performance i/o design patterns. http://www.artima.com/articles/io_design_patterns.html.

[7] ebson library at github. https://github.com/kohanyirobert/ebson.

[8] epoll(4) - linux man page. http://linux.die.net/man/4/epoll.

[9] epoll_wait(2) - linux man page. http://linux.die.net/man/2/epoll_wait.

[10] Event bus documentation of the akka project. http://doc.akka.io/docs/akka/2.0.2/scala/event-bus.html.

[11] Example of monitoring multiple stacks with valgrind. http://code.google.com/p/valgrind-variant/source/browse/trunk/valgrind/memcheck/tests/linux/stack_changes.c?r=114.

[12] Finagle api of the codec class located in the com.twitter.finagle package. http://twitter.github.com/finagle/api/finagle-core/com/twitter/finagle/Codec.html.

[13] Finagle, from twitter - network stack for the jvm that you can use to build asynchronous remote procedure call (rpc) clients and servers in java, scala, or any jvm-hosted language. http://twitter.github.com/finagle/.

[14] Finagle, from twitter - supported protocols. https://github.com/twitter/finagle/blob/master/README.md#Supported%20Protocols.

[15] Future documentation of the akka project. http://doc.akka.io/docs/akka/2.0.1/scala/futures.html.

[16] Gnu pth - the gnu portable threads. http://www.gnu.org/software/pth/.

[17] Gnu pth - the gnu portable threads - the world of threading. http://www.gnu.org/software/pth/pth-manual.html#the_world_of_threading.

[18] Greenlet project - lightweight in-process concurrent programming. http://pypi.python.org/pypi/greenlet/.

[19] Home page of twisted matrix labs. http://twistedmatrix.com/trac/.

[20] Homepage of the akka project. http://akka.io/.

[21] Homepage of the gevent project. http://www.gevent.org/.

[22] Homepage of the node.js project. http://nodejs.org/.

[23] Homepage of the stackless project. http://www.stackless.com.

[24] Java 6 io package api. http://docs.oracle.com/javase/6/docs/api/java/io/package-summary.html.

[25] Java 6 nio (new i/o) package api. http://docs.oracle.com/javase/6/docs/api/java/nio/package-summary.html.

[26] Java nio tutorial. http://tutorials.jenkov.com/java-nio/index.html.

[27] Jerkson - a scala wrapper for jackson which brings scala's ease-of-use to jackson's features. https://github.com/codahale/jerkson.

[28] Json in javascript. http://www.json.org/js.html.

[29] Jsonbox - this is a json c++ library. https://github.com/anhero/JsonBox.

[30] kill(2) - linux man page. http://linux.die.net/man/2/kill.

[31] Kqueue - freebsd man pages. http://www.freebsd.org/cgi/man.cgi?query=kqueue.

[32] libev - a full-featured and high-performance event loop that is loosely modelled after libevent, but without its limitations and bugs. http://software.schmorp.de/pkg/libev.html.

[33] libevent an event notification library. http://libevent.org/.

[34] Maximum number of threads and processs for windows xp. url-http://blogs.technet.com/b/markrussinovich/archive/2009/07/08/3261309.aspx.

[35] Messagepack - design of rpc - parallel pipelining. http://wiki.msgpack.org/display/MSGPACK/Design+of+RPC#DesignofRPC-ParallelPipelining.

[36] Messagepack project homepage. http://msgpack.org/.

[37] Mozzy mordor. http://code.mozy.com/projects/mordor/.

[38] Msdn - async (c# reference). http://msdn.microsoft.com/en-us/library/hh156513(v=vs.110).aspx.

[39] Msdn - asynchronous programming with async and await (c# and visual basic). http://msdn.microsoft.com/en-us/library/hh191443(v=vs.110).aspx.

[40] Netty - api - the replayingdecoder class. http://netty.io/docs/stable/api/org/jboss/netty/handler/codec/replay/ReplayingDecoder.html.

[41] Netty - documentation - interceptor chain pattern. http://netty.io/docs/stable/guide/html/#architecture.6.

[42] Netty - the java nio client server socket framework. http://www.jboss.org/netty/.

[43] The new *pselect* call. http://lwn.net/Articles/176911/.

[44] poll(2) - linux man pages. http://linux.die.net/man/2/poll.

[45] Protocol buffer basics: C++. https://developers.google.com/protocol-buffers/docs/cpptutorial.

[46] Protocol buffer basics: Java. https://developers.google.com/protocol-buffers/docs/javatutorial.

[47] Scala continuation and exception handling. http://stackoverflow.com/questions/6150331/scala-continuation-and-exception-handling.

[48] Scheduler documentation of the akka project. http://doc.akka.io/docs/akka/2.0.2/scala/scheduler.html.

[49] select(2) - linux man pages. http://linux.die.net/man/2/select.

[50] Serialization documentation of the akka project. http://doc.akka.io/docs/akka/2.0.2/scala/serialization.html.

[51] Stackoverflow - how to handle openssl ssl_error_want_read / want_write on non-blocking sockets. http://stackoverflow.com/questions/3952104/how-to-handle-openssl-ssl-error-want-read-want-write-on-non-blocking-sockets.

[52] std::shared_ptr - reference. http://en.cppreference.com/w/cpp/memory/shared_ptr.

[53] A taste of 2.8: Continuations. http://www.scala-lang.org/node/2096.

[54] Twisted class documentation for the protocol class. http://twistedmatrix.com/documents/11.0.0/api/twisted.internet.protocol.Protocol.html.

[55] Using openssl with asynchronous sockets. http://www.serverframework.com/asynchronousevents/2010/10/using-openssl-with-asynchronous-sockets.html.

[56] Valgrind project. http://valgrind.org/.

[57] Wikipedia article about event loops. http://en.wikipedia.org/wiki/Event_loop.

[58] Wikipedia article about fibers. http://en.wikipedia.org/wiki/Fiber_(computer_science).

[59] Wikipedia article about non-local jumps (setjmp.h). http://en.wikipedia.org/wiki/Longjmp.

[60] Wikipedia article about the monkey patch principle. http://en.wikipedia.org/wiki/Monkey_patch.

[61] Task dispatch and nonblocking io in scala, 2011.

[62] Dr. Frank B. Brokken. *C++ Annotations*. University of Groningen, version 8.2.0 edition, 2012.

[63] Richard Steiger Carl Hewitt, Peter Bishop. A universal modular actor formalism for artificial intelligence. 1973.

[64] Douglas C. Schmidt Thomas D. Jordan Irfan Pyarali, Tim Harrison. Proactor - an object behavioral pattern for demultiplexing and dispatching handlers for asynchronous events. *Pattern Languages of Programming*, 1997.

[65] Jim Mcbeath. Java nio and scala continuations. http://jim-mcbeath.blogspot.nl/2011/03/java-nio-and-scala-continuations.html, 2011.

[66] Eric Brewer Rob von Behren, Jeremy Condit. Why events are a bad idea (for high-concurrency servers). *HotOS IX Paper*, 2003.

[67] Tiark Rompf. *Node.scala* - implementing scalable async io using delimited continuations - presentation. http://days2011.scala-lang.org/node/138/288, 2011.

[68] Scott Rosenthal. Interrupts might seem basic, but many programmers still avoid them. http://www.sltf.com/articles/pein/pein9505.htm, 1995.

[69] Douglas C. Schmidt. Reactor - an object behavioral pattern for demultiplexing and dispatching handles for synchronous events. *Pattern Languages of Program Design*, 1995.

[70] Martin Odersky Tiark Rompf, Ingo Maier. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. *ICFP'09*, 2009.

[71] Paul Tyma. Thousands of threads and blocking i/o, the old way to write java servers is new again. `http://www.mailinator.com/tymaPaulMultithreaded.pdf`.

[72] Felix von Leitner. Scalable network programming presentation. the quest for a good web server (that survives slashdot). `http://bulk.fefe.de/scalable-networking.pdf`.