HAW HAMBURG

Bachelorarbeit

Jakob Otto

Redesigning and Evaluating the Network Stack in the C++ Actor Framework

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Jakob Otto

Redesigning and Evaluating the Network Stack in the C++ Actor Framework

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas C. Schmidt
Zweitgutachter: Prof. Dr. Franz Korf

Eingereicht am: 24. Juli 2020

Jakob Otto

Thema der Arbeit

Redesigning and Evaluating the Network Stack in the C++ Actor Framework

Stichworte

C++, Aktor modell, Verteilung, Netzwerkkommunikation

Kurzzusammenfassung

Die verteilte Programmierung stützt sich in hohem Maße auf die Kommunikation über das Internet, um die Skalierung und Verteilung von Aufgaben über physische Grenzen hinweg zu ermöglichen. Das Aktormodell erweitert dies, mit einem netzwerktransparenten Kommunikationsmodell unter Verwendung von Nachrichten. Das C++ Actor Framework (CAF) ist eine Implementierung des Aktormodells, das einen Netzwerk Stack Entwurf bereitstellt, der auf eine gründliche transparente Abstraktion der komplizierten Netzwerk-APIs abzielt. Es ist jedoch sowohl in der Erweiterbarkeit als auch in der Kombinierbarkeit begrenzt, was es schwierig macht, mit den sich stetig ändernden Anforderungen an einen solchen Stack Schritt zu halten. Neue Transportprotokolle wie Quick UDP Internet Connections (QUIC) oder Anwendungsprotokolle wie WebRTC wären gute Ergänzungen, die aufgrund des derzeitigen unflexiblen Designs nicht integriert werden können. Die Arbeit in dieser Arbeit überdenkt den derzeitigen Ansatz und schlägt ein neues Design für die Abstraktion der Netzwerkschicht vor. Dieser neue Ansatz wird in Bezug auf Zusammensetzbarkeit, Wiederverwendbarkeit und Erweiterbarkeit fertiggestellt, wodurch eine flexiblere Netzwerkabstraktion für CAF geschaffen wird. Eine Implementierung des neuen Entwurfs dient dazu, die Fähigkeiten und Grenzen aufzuzeigen, wodurch der Wert des Entwurfs für zukünftige Arbeiten bewertet werden kann.

Jakob Otto

Title of Thesis

Redesigning and Evaluating the Network Stack in the C++ Actor Framework

Keywords

C++, actor model, distribution, network communication

Abstract

Distributed programming relies heavily on communication over the Internet to enable the scaling and distribution of tasks across physical boundaries. The actuator model extends this by providing a network transparent communication model using messages. The C++ Actor Framework (CAF) is an implementation of the actor model that provides a network stack design that aims at a thorough transparent abstraction of the complicated network APIs. However, it is limited in both extensibility and composability, making it difficult to keep up with the ever-changing requirements for such a stack. New transport protocols such as Quick UDP Internet Connections (QUIC) or application protocols such as WebRTC would be good additions that cannot be integrated due to the current inflexible design. The work in this thesis reconsiders the current approach and proposes a new design for the abstraction of the network layer. This new approach will be finalized with respect to composability, reusability, and extensibility, thus creating a more flexible network abstraction for CAF. An implementation of the new design is used to show the capabilities and limitations, which allows assessing the value of the design for future work.

Contents

List of Figures

List of Tables

Abbreviations

ACE Adaptive Communication Environment.

BASP Binary Actor System Protocol.

CAF C++ Actor Framework.

FIFO First In - First Out.

HOL blocking Head-of-Line blocking.

HTTP Hypertext Transfer Protocol.

ICE The Interactive Connectivity Establishment.

ICMP Internet Control Message Protocol.

IETF Internet Engineering Task Force.

IP Internet Protocol.

IPC Inter-Process Communication.

MP Message Passing.

MPI Message Passing Interface.

MTU Maximum Transmission Unit.

NAT Network Address Translation.

OS operating system.

QUIC	Quick UDP Internet Connections.
SCTP	Stream Control Transmission Protocol.
TAPS	Architecture for Transport Services.
TCP	Transmission Control Protocol.
TLS	Transport Layer Security.
UDP	User Datagram Protocol.
UDT	UDP-based Data Transfer Protocol.
URI	Uniform Resource Identifier.

Listings

1 Introduction

Distributed Programming has gained much attention over the last decades. Companies like Google built large scale distributed systems to cope with the fluctuating load of their services. Distributed software brings difficulties that make designing and programming such systems very complicated [2]. Software like this need to be designed thoroughly and factor in reliability and scaling, as well as performance. Race conditions, the integrity of data, and partial failures of the system are just some problems that have to be kept in mind. Often programmers need expert knowledge of OS-specific mechanisms that can impact the performance greatly.

The actor model introduces a „share nothing“ approach, where actors cannot alter the state of another actor directly. The state of a specific actor can only be altered by the actor itself when a message is received, thus the actor model is thread-safe by default. To achieve this, message passing [3] is used as the only way of communication. This implies weak coupling between actors which in turn leads to very reliable and flexible software. Software systems consist of many interconnected parts, which are strongly coupled. Hence, the failure of single components can affect the whole system, which makes them error-prone. With the use of actors, such partial failures are unproblematic since the system is not coupled to the failing instance. Failed actors can easily be redeployed and continue where the failed actor has stopped.

Distributed computing relies heavily on the performance of the network and its abstraction, thus reliable networking capabilities are needed. Protocols like the Transmission Control Protocol (TCP) [4] or QUIC [5] offer ordering and loss detection, while simpler protocols such as User Datagram Protocol (UDP) [6] do not. Capabilities like ordering or loss detection are often desired. However, undesired features are often bundled with them. For this reason, composability can be very beneficial for building a reliable network stack. The possibility to add features to protocols enables programmers to build stacks specifically tailored for the use-case. Hence, the protocol can be chosen for its behavior instead of the reliability features it provides.

In this thesis, the current network abstraction of the C++ Actor Framework, as well as its limitations will be discussed. Furthermore, changes have been proposed that address these limitations, which will be evaluated.

1.1 Organization of work

Chapter 2 gives an overview over the actor model itself. In Chapter 3 problems with the current design are discussed, while related work is shown in Chapter 4. Chapter 5 explains certain design decisions while Chapter 6 discusses the actual implementation of the stack. The benchmark results are shown and evaluated in Chapter 7. Finally, Chapter 8 concludes the work and provides an outlook on possible future work in this field.

2 Actor Programming

The work of this thesis is based on CAF [1], thus the reader requires at least basic knowledge of actor programming and its benefits. This chapter provides a short overview of the actor model of computation and introduces CAF and some of its features.

2.1 The Actor Model of Computation

Against the background of the ever-growing software demands, computing performance needs to be improved. In the past, the solution for this was building larger-, more powerful computers which proved to be very costly and time-intensive. A more modern approach is to distribute software systems across many nodes. This way, the need for single, high-performing supercomputers no longer exists. A more recent approach is to build distributed systems, which are scattered across the world. This approach is much cheaper, while the performance can be as good or better than the conventional approach. Furthermore, the scalability of such systems is way better compared to the approach using undistributed computer-systems.

First proposed by Carl Hewitt et al. in 1973 the actor model was intended for the use with AI [7]. In 1986 Gul Agha has formalized this idea [8], which has then become a more general programming paradigm. „Actors are concurrent, isolated entities that interact via message passing“ [3]. Actors consist of only a behavior, which describes *how* they react to messages, their state, and a mailbox for incoming messages. Each actor consumes messages sequentially from its mailbox until it is empty. Every time a message is consumed, a matching message handler from their behavior is called. This strict sequential order of events is thread-safe by default and thus synchronization steps can be omitted.

Actors are identified by unique identifiers, which enables transparent addressing of individual actors. Because of this, messages can be dispatched transparently to either a local or a remote actor. This feature enables porting such software from a local multi-threaded to a distributed context easily with very few changes.

Upon receiving a message, actors can react in three distinct ways:

1. Send messages to other actors
2. Alter its state (including the behavior)
3. Spawn new actors

This very restricted set of actions avoids common bugs like race conditions or data corruption.

The first implementation of the Actor Model has been proposed with Erlang in the mid-'80s. Even though the creators of Erlang never actually referenced the Actor Model, it still is very close to the ideas of it. The main goal was to create a language for use in the field of telecommunication technologies. Key requirements were fault-resistance and distribution–The resulting software should be able to „run forever“ [9].

In the last decade, other implementations of the Actor Model have emerged. Frameworks like Akka [10], Orleans [11], or CAF [12] are only some examples.

2.2 C++ Actor Framework

CAF is an implementation of the actor model written in C++. The framework has many advantages when compared to other implementations of the actor model. Software that is written in C++ is compiled ahead of time for the targeted architecture and operating system (OS). Thus, it does not rely on a virtual machine and runs natively on almost any hardware.

Two different approaches to implement actors are provided: function-based (using lambdas), or class-based. Programmers can thus choose between a strictly functional or an object-oriented approach. Furthermore, spawning and deleting actors are both efficient operations, since CAF actors have a very small memory impact. This enables to rapidly spawn and kill actors even for very small tasks, without adding a noticeable impact to the runtime.

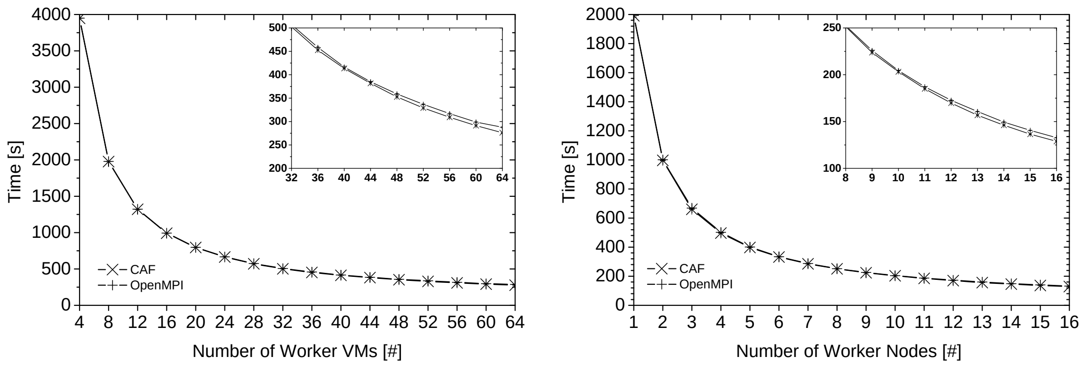


Figure 2.1: Comparison between CAF and OpenMPI [1]

Actors are scheduled by the `actor-scheduler`, and executed by `workers`. `Workers` are thread abstractions designed to execute the lambdas provided by actors. This approach enables rapid switching between many actors without dealing with the overhead of scheduling many threads. When an actor needs to call slow or blocking system calls, actors can be detached. A detached actor is executed by its own thread and thus, does not rely on being executed by a `worker`. Hence, it cannot block the execution of other actors.

Another feature provided by CAF are statically-typed actors. Such actors enable the compiler to check the messaging interface against all incoming messages. This feature is particularly interesting in a distributed setting because actors are allowed to drop messages they cannot handle without an error message. This circumstance makes finding such bugs very difficult since there might be no debugging information present. To use statically-typed actors, remote actor handles have to be converted to the type of the remote actor before sending messages to it. While this allows type-safe communication between actors which eradicates a whole error category, it introduces only a small implementational overhead. However, since the type checking is done at compile-time, no runtime overhead is added.

The memory layout of actors in CAF has been optimized to make perfect use of CPU cache lines. The actor base-class has been carefully padded to fit into a single 64 byte CPU cache line which is the most common size in recent CPUs. This prevents an issue called false sharing [13], where two CPU cores would interfere with each other when accessing the CPU cache.

Since message passing is a key part of the actor model, CAF provides a unique message passing implementation. The message passing layer features typed interfaces, and thus the ability to use typed messaging, which is unusual for Message Passing (MP) implementations. Since commonly used implementations of MP are written in `C`, type information is often hidden behind void pointers. Charousset et al. [1] have compared OpenMPI [14] and the implementation in CAF, by sending and processing images of the Mandelbrot set in a distributed system. As shown in Fig. 2.1, they found that the two implementations have almost identical runtimes.

3 Problem statement

CAF features a network abstraction layer that is specially designed for actor messaging and the management of CAF nodes. The implementation has grown over the years and requirements for it have changed, thus it does not meet the expectations anymore. This chapter explains the requirements for such a network abstraction layer and proposes changes that target issues with the current implementation.

3.1 Configurability

A crucial step when designing networking capable software, is to select the best fitting transport protocol for the use-case. The most widely used transport protocol is TCP [4], which bundles reliability guarantees such as retransmission, congestion control, and more. These features allow transmitting large quantities of data over the Internet without the risk of loss or data corruption.

The current network design in CAF is strongly coupled to TCP, because the protocol reduces the implementation overhead. Designing and implementing reliability features is a time-consuming and complex task, which is not necessary when using a reliable protocol. Since the field of application of CAF had no use for other protocols at the time, the overhead was left aside until such use-cases would emerge.

3.1.1 Transport Protocols

TCP is the current de facto standard because of its many benefits, but also has issues that should be known. Managing large numbers of TCP connections, for example, adds significant overhead to the runtime and memory usage [15]. The retransmission feature uses timers to trigger retransmits when packets have been lost. Since communication over TCP is ordered, all packets that have already been received are held back, until the missing packet(s) have arrived. The added memory overhead comes from the additional state that has to be held for the connection itself. Every packet that has been sent, has to be held and tracked until the remote node acknowledges the transmission. While the memory usage per connection is limited, this adds up with a growing number of sockets and can impact the performance of resource-limited systems.

In CAF, each actor that should be accessible over the network, currently adds a socket to the runtime. This potentially increases the number of sockets that have to be managed by the CAF runtime significantly. Such behavior is comparable to a busy web server, that needs to manage many connections simultaneously. Thus, this situation could impair the performance of resulting software written using CAF and should be kept in mind when designing a network topology.

Another issue with TCP is the Head-of-Line blocking (HOL blocking), which can occur in switching hardware. This kind of hardware is commonly implemented using a First In - First Out (FIFO) buffer per port, that forwards incoming packets in the same order as they were received. Since IP packets are routed individually, a previously ordered stream of packets can be received out of order and on different ports of the device. In the case of congestion on a route to a destination, the packets to that node are withheld, which blocks all other packets in the queue. This behavior can limit the throughput of such a device to up to 58.6% [16]. A possible solution for this problem is to use virtual output queuing instead of simple FIFO buffers. By virtualizing the buffers for each endpoint, HOL blocking can be solved, however, this implies that a sending endpoint can not avoid such a situation if it is connected to a switch without this capability.

A quite critical issue with TCP is that it is not appropriate for the use in real-time applications such as Voice over IP (VoIP) or gaming. Due to the situation with conservative timeouts, the runtime behavior is not sufficient for such use-cases. It adds significant overhead to the application, which has to wait for data that is possibly outdated by the time the missing packet has arrived. Simpler protocols such as UDP or more specialized ones like Stream Control Transmission Protocol (SCTP) would be more appropriate choices for this field.

Currently, the networking capabilities of CAF are very restricted due to the strong coupling to TCP. The design of the stack has been implemented in a way that switching protocols, implies many changes in the overall implementation. Thus, configurability options for including different protocols easily has not been included in the design. This limits the application domains in which CAF could be used. Also, the overall design has been carefully designed and optimized in terms of performance–But only with TCP in mind. Due to this, the level of complexity has risen to a level where a redesign is necessary to clean up the code to make it easier to maintain.

3.1.2 Application Layer Protocols

Application layer protocols are situated on top of the transport layer and responsible for processing and preparing data. Usually, they represent a software-specific behavior, that is used to communicate between different parts of software running on individual nodes. Well known examples for application protocols are Telnet [17], HTTP [18], or SSH [19], which are all implemented in the application layer.

The ability to use and implement such protocol abstractions is crucial in the process of building a reliable network stack. Reliability options can be added and stacked with this kind of approach. This would be an important addition because it enables programmers to add such options to more limited protocols such as UDP to improve performance.

The current design in CAF does not provide an API to implement application protocols to the stack. However, they can still be included by implementing `brokers`, which are a special kind of actor. These entities manage the way a specific endpoint behaves while communicating over the Internet. To do this, brokers provide a set of message handlers to process incoming `raw_data_messages` that are then passed on to the receiver. The best example for this approach is the Binary Actor System Protocol (BASP) broker, which handles all communications between different CAF nodes, including `actor-messages`. This approach is scalable and fits perfectly into the actor communication API (because brokers are actors).

However, this solution also has limitations that are impractical for the end-user of CAF: The modularity. More specifically, brokers can not be stacked or chained together to form a protocol stack. Using multiple application layer protocols together is difficult and tedious because possibilities to compose brokers are limited. The only way of accomplishing this behavior is to implement a single broker that handles both protocols at the same time, rendering modularity virtually impossible in this design.

Additionally, the lack of possibilities to add certain reliability options to protocols leads to strong coupling to the guarantees of the used protocol. Since CAF currently relies solely on the guarantees of TCP, the error propagation of it is limited. For example, determining the liveliness of another CAF node is solely accomplished by the state of the connection to it. This is especially critical in the case of frozen nodes. Such a state does not imply that the connection to the stale node is closed, thus the connection state does not reliably propagate the state of the remote node. Hence, a full rewrite of this abstraction in favor of a configurable design is necessary to add reliability options such as gossip protocols [20] to determine the state of remote nodes.

3.2 Performance of the Network Stack

Along with the reliability, another key goal of communication over the Internet is the performance. Such performance is measured by the transmission latency, the throughput, and jitter. Latency describes the duration between the arrival of packets–If this interval is high, throughput and reactivity of systems is affected. Continuously changing latency is called jitter, high amounts of jitter can lead to congestion, which in terms can decrease the overall network throughput drastically. Lastly, throughput is the total amount of data that can be transmitted over the network. Naturally, high throughput networking is the goal of any program that is using the network to communicate.

While the performance of a given network cannot be affected directly by the software, it can aim to exhaust the networking capabilities of the OS it is running on. By limiting the added overhead of the network abstraction, for example, a significant performance loss can be mitigated. The efficient processing of data can increase throughput significantly because it allows the OS to send the data in a continuous stream. The current implementation of the network stack in CAF has not met this requirement, due to inefficiencies with serializing and multiplexing.

3.2.1 Serializing

Transmitting data over the Internet poses the problem of correct data representation. The simplest example is endianness: The byte order within the binary representation of an integer number. To avoid this problem, data must be serialized into a previously defined format.

The process of serializing includes copying the data that should be sent into another buffer, which is an expensive task. A possibility to limit the cost of this is to parallelize the work. Such a solution requires dividing the data into chunks, which can then be processed concurrently by multiple serializers. While this approach could potentially deal with the overhead of copying, other overheads, such as synchronization or scheduling are added and have to be closely watched.

The current implementation in CAF prevents this since it was designed with the task being strictly single-threaded, hence the current serialization process scales poorly. Message buffers for serializing into, as well as the unique message queues of the socket-managers, have not been implemented for such a use-case and thus lack synchronization. On the other hand, deserialization is already implemented concurrently, which improves performance noticeably.

Thus, serializing data concurrently is a solution to mitigate the impact of copying. It can reduce the resulting waiting time of socket-managers, which will improve the overall performance and reactivity of resulting software.

3.2.2 Multiplexing

Managing large numbers of connections is a costly task, thus using asynchronous I/O and efficient event multiplexing is important. OS features, such as `select`, `poll`, and `epoll` are implementations of the reactor pattern, which is designed to handle this task. The pattern monitors a set of sockets for events and triggers specific events when reading or writing is possible on a socket.

Fig. 3.1 illustrates the process of multiplexing I/O events. Necessary components are a reactor implementation such as `epoll` and a set of event handlers. Handlers are coupled to an open socket and added to a pollset which the reactor monitors. Each time an event on one of the sockets occurs, the corresponding handler is triggered with the type of event (read, write). The event handler then proceeds to handle the event by writing to or reading from its socket. When data is received it forwards it to a sink, which can then process it.

Since this routine is commonly used in asynchronous communication, the current implementation of CAF relies on `epoll` as reactor implementation. Event handlers are implemented in the form of the `stream` class, while the `broker` class functions as a sink for the data.

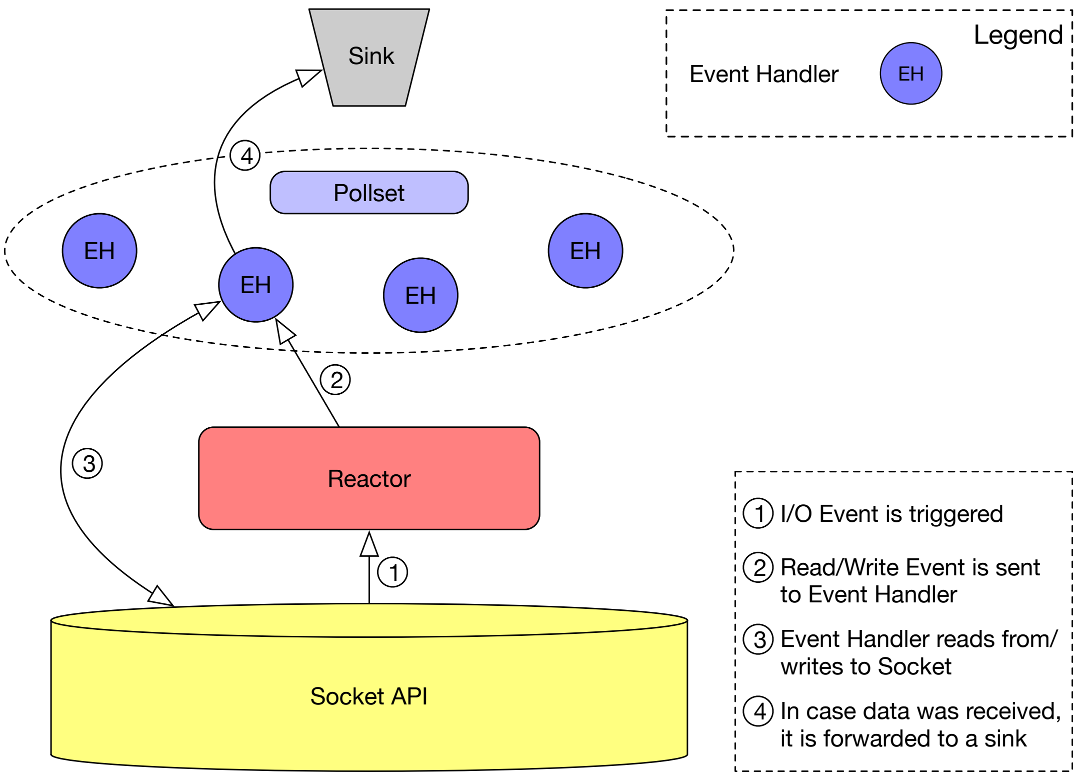

Sink
Legend
Event Handler
EH
4
Pollset
EH
EH
EH
EH
2
3
Reactor
1
Socket API
1 I/O Event is triggered
2 Read/Write Event is sent to Event Handler
3 Event Handler reads from/ writes to Socket
4 In case data was received, it is forwarded to a sink

Figure 3.1: Generalized overview of the multiplexing process

The multiplexing process is currently implemented strictly single-threaded, which limits the capabilities of the multiplexing design. Parallelizing this routine could potentially add a substantial performance gain to the network stack implementation. Multiple multiplexing instances could handle connections concurrently, thus distributing the overhead of managing ever-growing poll sets across them.

Thus, a new thread-safe design for the multiplexing implementation is proposed. This would open the possibility of running multiple `multiplexer` instances concurrently, thus enhancing the performance of the whole network stack.

3.3 Resolving Actors

To be able to offer their service, actors have to be accessible over the Internet. Thus, publishing and resolving actors are essential processes when building distributed systems. CAF currently implements this process by assigning a unique socket to every actor that needs to be accessed remotely. It is then published by binding its socket to a port, which allows accepting incoming connections from remote nodes. Accessing such an actor can then be done by connecting and carrying out a `basp-handshake`. To minimize the number of open sockets, the runtime then checks whether the remote node is already known or not. If that is the case, the newly established connection is dropped, and the communication channel is moved over to the already existing connection.

While this approach is easy to implement, it is laborious and also bears some disadvantages. Opening a connection to an already known CAF node simply to do a handshake and close it afterward is unnecessarily complex and adds a significant runtime overhead to the connection process. Since the host is known beforehand, a simpler approach would be to check for an existing connection beforehand. Publishing single actors to ports, also renders it impossible to identify the type of actor without connecting to it. Currently, a handshake between both nodes is required to figure out the type of the actor. Furthermore, the resolving process lacks proper error propagation. Only errors of the underlying transport protocol are observed, which limits the feedback of the process to connection problems. This type of error is inexpressive in this situation because transport problems are just a subset of the possible reasons the resolving of actors can fail.

3.4 Encrypting Network Communications

Data is often sent over the Internet in the form of plain text and is thus readable by anyone who obtains a copy of it. This is undesirable, especially when sensitive data such as personal information or passwords are transmitted. Hence, encryption is often used to secure such data so that only authorized participants can decipher, and read it.

Two main communication models can be implemented when using encryption: The client-server model or the end-to-end model. The client-server model encrypts data only from host to host, thus the data is only encrypted between two nodes. This implies that the data is only secure when communicating directly with the receiver. End-to-end encryption is a variant of this model, which encrypts the communication between the sender and the actual receiver of the data. Thus allowing data to be securely forwarded, even when multiple hops are necessary to reach the receiver.

Since actors are extremely volatile by design, the end-to-end model is not suitable for the actor model. Encryption contexts would have to be created and handshakes have to be carried out for every new actor that is spawned, which adds an unnecessary overhead to the communication. The client-server model is much better suited for this use-case, since CAF nodes are not as volatile as actors.

Thus, encryption is a very usable feature, that should be implemented in a feature-rich network stack abstraction. CAF currently provides `libcaf_openssl`, which relies on the OpenSSL project [21] for the actual encryption of data. It uses the client-server model to allow secure communication between different CAF nodes. However, the library in CAF is still in experimental state, and uses the broker application protocol abstraction design. An improvement would be to implement this using a configurable design to enable encrypted communication for other protocols as well. For example, OSCORE [22], which is currently being drafted by the Internet Engineering Task Force (IETF), would be a valuable addition.

3.5 Overlay Networking

Overlay networking is a technique that is used to build a network on top of an already existing network. It can be used to improve the reliability of a network by providing multiple communication paths, and routing data accordingly.

CAF currently implements facilities to create such overlay networks by exchanging routing tables between CAF nodes. With this, CAF nodes do not have to maintain connections to every single actor. They only have to know which node can forward messages to the requested endpoint. Thus, actors can communicate using already existing channels by sending messages that are forwarded by other CAF nodes. This feature is especially helpful in the case of unreliable connections, enabling the use of the established overlay in case of connection failures.

However, while this approach can improve overall reliability, it also adds a great deal of complexity and management overhead to the network layer. Making educated decisions about how to route specific packets is a very complex task [23], which has already been implemented by the Internet Protocol (IP) [24]. Another downside is reliability in the implementation of CAF, which does not propagate errors from remote nodes. This can lead to the silent dropping of messages that cannot be delivered.

Since the network stack in CAF does not benefit much from overlay networking, the feature should be completely removed from the new design. Thus, removing the management overhead as well as simplifying the code of the implementation.

4 Related Work

This chapter covers related implementations and topics from the field of network abstraction design. It begins with an overview of communicating in the field of parallel computing, covering IPC and different approaches for it. After that, previous efforts of redesigning the network layer in CAF are explained and discussed, before going over several other frameworks and their implementations.

4.1 Inter-Process Communication

Distributed computing is a special case of parallel computing. Both paradigms are widely used in the field of high-performance software. They allow dividing large tasks into smaller problems, which can then be executed concurrently. This defines the field of application for computing clusters that scale from a single machine up to highly distributed systems with many nodes. Managing such systems opens the demand for reliable ways of communication between processes to notify and share data between them. This is called Inter-Process Communication (IPC) and is often implemented using OS primitives such as shared memory, signals, or pipes when it is done locally. In the field of distributed software, the only feasible way of communication is the Internet by using sockets.

4.1.1 OS Primitives for IPC

IPC can be implemented using a variety of approaches. The most commonly used approaches are explained briefly in the following section:

Signals The concept of using signals is one of the oldest paradigms and has been implemented in virtually all OSs including Windows. While this idea is very simple and easy to use, it is also very limited since it can only be used to notify processes or trigger tasks. Signal handlers have to be implemented, which can only receive a specific signal to trigger them–Implying that transmitting data is not possible.

Shared Memory Shared memory is a very commonly used approach, where two or more processes use the same memory region to communicate. This approach is very performant, but since multiple processes access the same memory concurrently, the integrity of the data needs to be ensured. Thus, requiring expert knowledge about synchronization facilities like `mutex` or `semaphore` to implement thread-safe access. It is very easy to introduce an unnecessary performance overhead to such a solution, which can be mitigated by identifying critical sections and keeping them as small as possible. Hence, the resulting code often leads to race conditions or deadlocks which are hard to debug because of the complexity that is added by synchronizing.

Pipes Pipes are simple FIFO buffers that reroute the output-stream of a given process to the input-stream of another. They are a fundamental part of Unix-like systems and can be used to send data or just trigger events. Two different types of pipes exist, named-, and anonymous pipes. Named pipes can be created by one, and obtained by another process using the identifier of it. Anonymous pipes are usually used in shell-environments to pipeline different programs together.

Sockets Sockets are a network communication abstraction, that can be used to communicate locally (Unix Domain Sockets) or over the Internet (Internet sockets). This abstraction can either be datagram- or stream-oriented, with a variety of underlying protocols, such as TCP or UDP. However, using socket communication locally is not as performant as shared-memory for example, thus it is not commonly used for local communication. Communication over the network, on the other hand, can only be implemented using this approach since it is the de-facto standard to realize such communications. Hence, this approach is used to communicate in distributed software systems.

Most of these primitives are suited very well to communicate locally between processes. However, the problem of communicating between remote nodes can only be solved by, or in combination with sockets. As an example, distributed shared memory has found application in distributed software, which is a combination of sockets and shared memory. This approach relies on a central node that provides memory that can be accessed over the Internet.

Frameworks such as OpenMP [25] provide support for implementing this approach. The whole paradigm of distributing this approach has difficulties with proper scaling though. In a local environment on just a single node, the overhead that is added by synchronizing is manageable because once accessed, the copying of data is comparatively fast. Copying data across different nodes over the Internet though usually is not. Since in a distributed system the shared memory has to be accessed using this way, copying the data takes significantly longer. Thus, processes have to wait longer until they can enter the critical section, which leads to an overhead that can drastically affect the performance of the whole system.

Another approach is to use plain socket communication. However, this requires at least a simple protocol that defines a set of rules for the communication process. Thus, a basic network stack is needed to realize this solution, which makes it more complex and time-intensive to design and implement.

4.2 Network Stack Abstraction

Building network abstraction with high performance can get complicated very fast due to the many performance-limiting factors that have to be kept in mind. Scaling of the stack and processing the data are common pitfalls that lead to high latency when the number of connections grows.

Thus, a common way to ease the process is to implement it in a stacked approach. In such an approach, the functionality of the abstraction is divided into different pieces, which are then implemented as protocols. Each protocol provides very limited but specific functionality, such as data representation, data transmission, or reliability. These protocols are then layered to form a network stack, where each layer can only interact with the two adjacent layers directly.

Since communication over the Internet is inherently unreliable, one of the central elements when building such protocol stacks is reliability. Often, reliability options are inherited from using sophisticated protocols like TCP or SCTP, which are reliable by design. However, because of this, they can add a significant runtime and memory overhead to the stack, which makes them inappropriate for embedded or real-time applications. A more appropriate solution for such use-cases would be to use simpler protocols such as UDP and adding the required guarantees in the form of application protocols. Including configurability options would widen the application domains of such a stack, rendering it a necessity for any sophisticated implementation.

4.2.1 Configurable Networking in CAF

The idea of redesigning the network layer in CAF has been discussed before by Hiesgen et al. [26, 27]. Their work was focused on topics such as reliable communication, reachability of actors, and scalability. This lead to a partial redesign that added configurability to the new stack to address the discussed topics.

Configurability options in CAF are very limited because it has been designed only with TCP in mind. The strong coupling is due to many benefits that came with relying on TCP as the only transport protocol. For example, CAF did not have to bundle a custom reliability layer because relying on the guarantees of the protocol was sufficient. Liveliness indicators could be derived from the state of the connection between remote nodes, while ordering and loss detection are implemented in TCP itself.

Apart from reliability and messaging guarantees, the reachability of actors was another field of discussion. When building distributed systems using actors, they have to be reachable and thus addressable over the Internet. The current implementation addresses this requirement by binding certain actors to ports, which can be accessed this way. However, due to firewalls or Network Address Translation (NAT), direct addressing of such actors is not always possible. Hence, Hiesgen discussed the use of The Interactive Connectivity Establishment (ICE) [28] in more depth, which can be used to establish connectivity for offer-answer protocols behind NATs and firewalls.

Another topic was the scalability and performance of the current network design. CAF can run using an arbitrary number of nodes, which may communicate with each other. This can result in degrading performance due to the growing number of connections that have to be managed by the networking layer. The very high level of abstraction that CAF offers adds another significant overhead to the implementation. Processing previously received data is a costly task that can be done in a reasonable time. This includes parsing message headers, serializing and deserializing data, and delivering messages to the receivers. However, when the number of different endpoints grows, the network stack has to be able to scale with the growing load. This is a problem that has been pointed out in their work and should be closely watched in any redesigning efforts.

The discussion of the current situation in the network stack implementation resulted in a partial redesign. A crucial modification was the new configurable design which allowed application protocols to not be implemented as `brokers` anymore. This step enabled users to build network stacks that could consist of an arbitrary number of layers. Transport protocols of choice could be included, which could now be extended with (custom) application protocols.

The new design also featured a rework of the `event_handlers`, which were then implemented to be actor-based. This change was thought out to increase the scaling capabilities of the network layer while simplifying the scheduling process. CAF provides a sophisticated work-stealing `actor_scheduler` that could be used for this task. While this idea seemed promising, it proved to add unnecessary overhead to the event-handling process.

However, a new protocol called QUIC [5] could not be added to the stack, despite the redesign. It is stream-oriented but relies on UDP as the underlying transport protocol. Due to the API of many of the reference implementations for QUIC, it was neither handled like a stream-oriented nor a datagram-oriented protocol. This approach has shown that there would be problems with other protocols as well.

Hence, the redesign has not been included in CAF but yielded a lot of insight and concepts on how to create a better-suited design. The three most important ones were:

- The transport abstraction should be as generalized as possible.
- The new design should to be configurable.
- Multiplexing should be thread-safe for future performance improvements.

4.2.2 Netty

„Netty is an asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers & clients“[1]. Netty is a networking framework that is very commonly used in Java projects that need to communicate over the Internet. It features commonly used transport protocols such as UDP and TCP, as well as SCTP and UDP-based Data Transfer Protocol (UDT) which are

[1] `https://github.com/netty/netty`

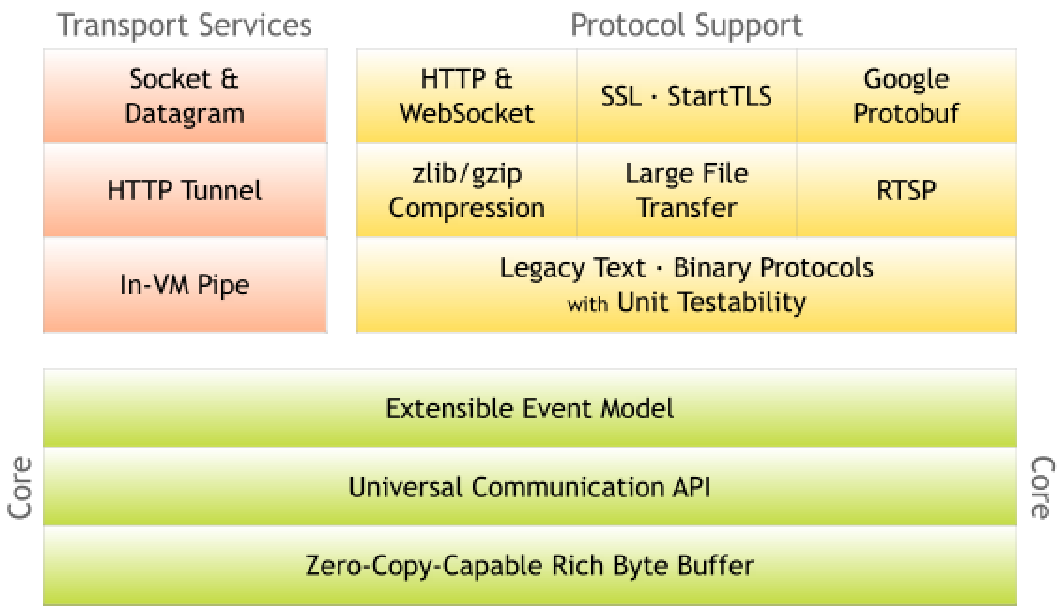


Figure 4.1: Overview of the components in Netty[2]

more specialized and not as widespread. Also featured in the framework is native support for the inclusion of application layer protocols such as HTTP(2), SMTP, and many more.

Design Decisions

Fig. 4.1 gives an overview of the components that Netty provides to build applications with. Especially noticeable in the core framework is the extensible event model, which includes reactors and the universal communication API. These components are necessities for implementing scalable and high performing networking applications. The shown transport services are just an excerpt of the provided protocols, but especially socket and datagram abstractions are basics in network programming. Also included is a variety of application protocols, which include encryption, compression, WebSockets, and many more.

Netty relies on an event-driven design, based on the reactor pattern using either `poll` or `epoll` from the Java core NIO library. This allows reacting upon I/O-events, rather

[2]`https://netty.io/`

than polling for them on every single open channel. By monitoring the set of Java streams, it can trigger such events when a stream is ready to be read from or written to. Thus, making networking using Netty very responsive and efficient, while being very scalable.

To achieve a high level of abstraction, Netty relies on a design using the so-called `Channel` and `Handler` classes to build a stack abstraction. `Channels` are asynchronous, event-driven abstractions of transport protocols, which offer a generalized API. Common actions, such as `bind`, `close`, `read`, or `write` are provided by this interface. When using stream-oriented protocols, a single channel represents a connection to another node, much like the `Stream` approach in Java. With the usage of datagram-oriented protocols, `channels` are comparable to unconnected sockets, since no connection is established for this type of protocol.

The application protocol abstraction is provided in the form of so-called `Handlers`. These specify the behavior of the protocols while encapsulating the specifics, which allows stacking them. `Handlers` can easily be added to a `Channel`, to processes (and act upon) the received data before returning it to the user. This approach follows the interceptor pattern, which is based on injecting an interceptor in between two stack layers. Surrounding layers do not need to know about the interceptor, hence this approach is transparent. It also offers a very high level of abstraction by separating concerns into separate classes while minimizing the implementational overhead.

However, the process of implementing multiple layers of protocols is only possible by inheritance. Protocols have to derive from the following ones and forward the processed content to them by calling `super.messageReceived(ctx, <content>)`. This leads to a closer coupling of the different layers, which should not necessary with such a design.

Discussion

Netty offers a very high level of abstraction by providing various wrappers for the low-level Java API. Even with this kind of overlay, it is possible to use a variety of different transport protocols which makes the framework versatile and attractive for many use-cases. Managing a vast amount of connections at the same time is implemented very efficiently by using event-multiplexing. This approach eradicates the need for expensive polling strategies and unnecessary blocking calls. Thus, the framework can scale very well with a growing number of streams, while the performance is not degraded.

The design of the library is sophisticated and very versatile, which makes building complex network stacks easy and fast. Including custom application protocols is possible, simple, and thus a key argument for this library. After all, when implementing software with networking capabilities in Java, Netty is the go-to solution for almost any use-case there is.

The network abstraction in CAF should include an application abstraction modeled on the design of Netty. Since their design allows implementing protocols as distinct entities, the modeling process is very simple. Furthermore, including the protocols in the stack is very simple and user-friendly. The interceptor-based inclusion of the design in Netty is versatile and thus very extensible. Thus, the new design in CAF should implement the inclusion similarly by using this pattern.

4.2.3 Boost Asio

Boost Asynchronous I/O (Boost Asio) [3] is a framework that provides a high-performance asynchronous I/O abstraction in C++. Originally, the focus was to provide such capabilities only for socket communication. However, the library now supports many more, including serial-ports and file descriptors. It is focussed on forming a high-level abstraction for the low-level `C` socket API, which is known to be very complex.

Design Decisions

As shown in Fig. 4.2, Asio provides several building blocks to build network stacks from. Components such as sockets, resolving facilities and TCP/UDP support are provided. Also featured are high-precision timers for triggering events and SSL/TLS support for encrypting data. Since the library supports a variety of OSs, Windows and POSIX conformal APIs are implemented.

A high level of abstraction is achieved by implementing wrapper types for the basic OS components. These offer an API for accessing and calling system functions to achieve connectivity, or send/receive data from the network. Two components are essential in the Asio design: The `io_context` and `io_object` classes. `io_objects` function as wrapping types for OS resources, such as sockets, file-descriptors and the like. This allows

[3]`https://think-async.com/Asio/asio-1.16.0/doc/`

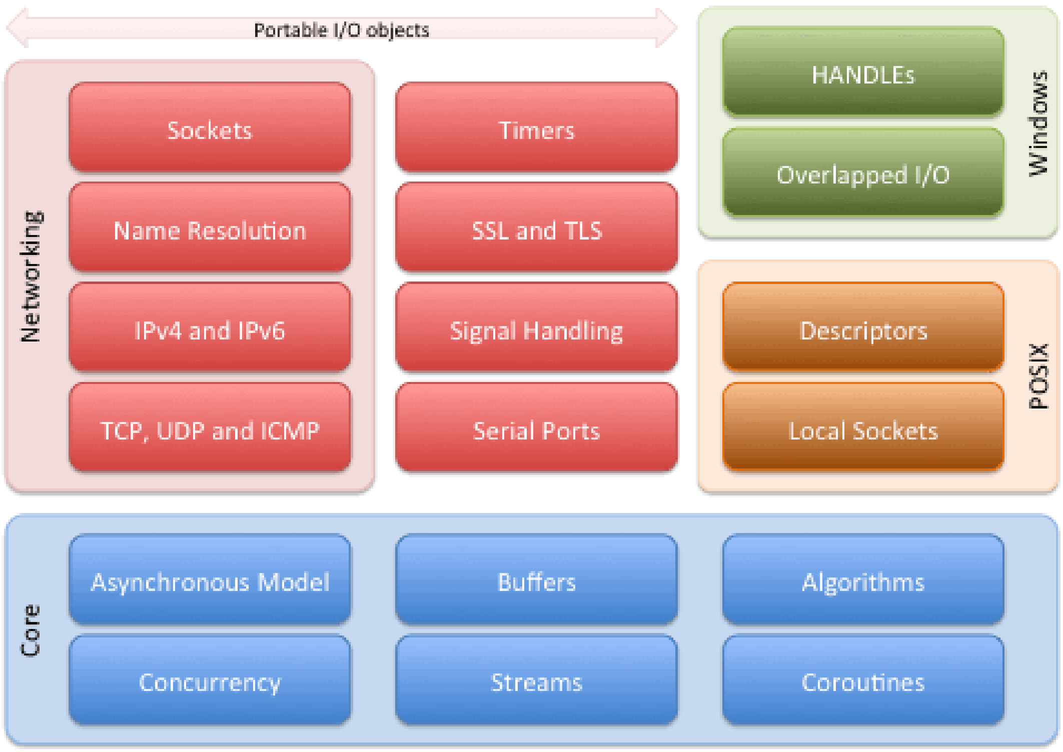


Figure 4.2: Overview of the basic building blocks of Asio[4]

the library to offer support for a variety of different I/O channels. The `io_context` class wraps the actual system calls that have to be performed.

While the framework itself aims to offer simpler implementations of asynchronous communication, it can also be used to perform synchronous I/O. The library provides an extensive timer API, which is used as a base for any actions that should be performed. I/O tasks can be scheduled by blocking on a `timer.wait()`, which can be resource-intensive. Thus, another approach is to use functions or lambdas that are passed to the timer, which will execute them on a timeout. This design enables programmers to follow either a thread-based or a timer-based approach, which makes the framework very versatile.

Asio provides support for three transport protocols: UDP, TCP, and Internet Control Message Protocol (ICMP). This allows using the framework in a variety of fields such as

[4]`https://think-async.com/Asio/`

embedded or high-performance computing. However, since the abstraction provided is based on the socket API, using other protocols is not possible, which limits the versatility. Furthermore, the framework is intended to function as an abstraction for asynchronous I/O and thus does not offer any way to include application protocols. This limits its field of application to the transport layer of applications which implement necessary protocols on top of this stack.

Discussion

Asio's extensive abstraction provides components for almost any use-case in networking. A reactor pattern implementation based on `poll`, `epoll`, or `kqueue` is included. This offers event multiplexing capabilities that enable managing sockets using socket-events. This approach is very scalable and limits the performance impact added by large pollsets.

The approach of providing a limited but simple abstraction for networking is helpful when designing and implementing network stacks. However, due to the lack of application protocol support, building complete network stacks on top of this library is still a challenge. Since the data that is obtained by it is still in raw, unprocessed form, all the parsing and handling have to be implemented on upper layers. Furthermore, the number of transport protocols is rather limited, since only TCP, UDP, and ICMP are supported.

After all, the framework is intended only as a transport layer abstraction and achieves this goal very efficiently for common transports.

4.2.4 The Adaptive Communication Environment

The Adaptive Communication Environment (ACE) [29] is a framework for building concurrent communication software in C++. It is well known in the field of high-performance distributed computing because of its reliability. Work on this library has been ongoing since 1993, thus the implementation is very sophisticated. Many performance- and safety-critical projects rely on ACE as their communication backend[5].

[5]`https://www.dre.vanderbilt.edu/~schmidt/ACE-users.html`

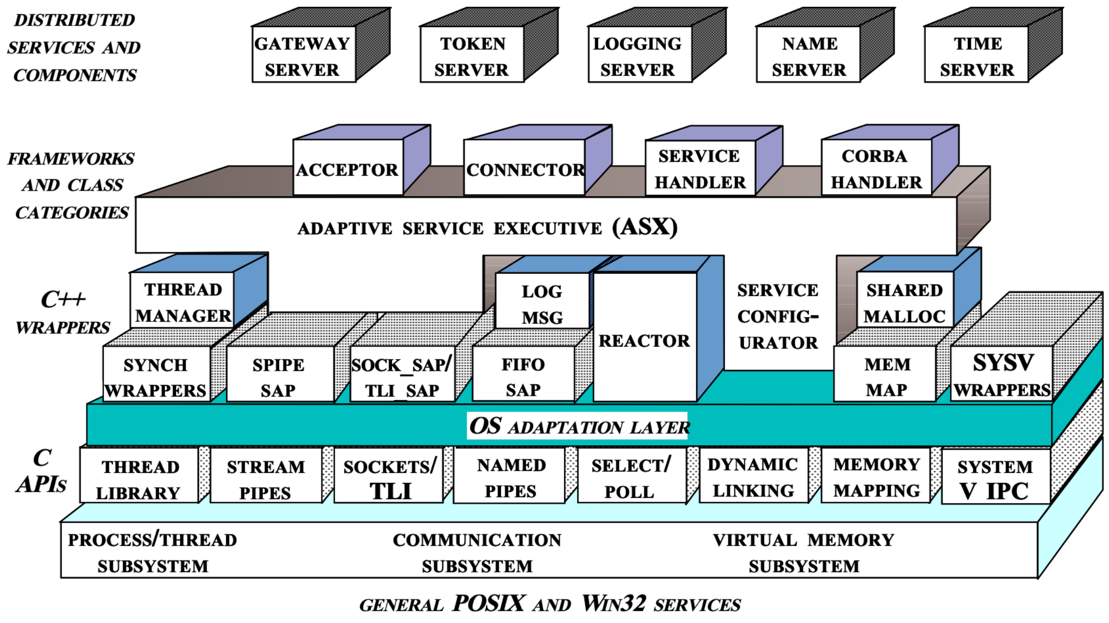


Figure 4.3: Overview over the ACE design[6]

Design Decisions

Fig. 4.3 provides an overview over the design that is used in ACE. The design is separated into multiple layers the first layer creates a platform-independent abstraction over the various networking APIs. This is done by providing an OS adaption layer, which functions as a base for the framework itself. The framework is built in a very modular way, which allows programmers to use only selected parts of it. For example, relying solely on the OS adaption layer to build network-enabled software in a low-level but platform-independent approach is possible. However, the following layer features a high-level abstraction by offering various wrappers that expand and improve it.

The set of wrappers provides consistent access to system functions for implementing IPC, multi-threading, synchronization, and more. They are written in C++, thus following a strictly object-oriented design, which adds valuable type-safety to the implementation. Components for event demultiplexing, service initialization and configuration, and streaming capabilities are provided. The event demultiplexing components include reactor and proactor implementations that can be used to dispatch a variety of events

[6]https://www.dre.vanderbilt.edu/~schmidt/ACE-overview.html

including I/O, timers, and signals. This expands the versatility and scalability of ACE, which thus can be used for a variety of applications.

The actual framework is sitting on top of this layer and includes a connector, acceptor, and handler implementations. Connectors and acceptors simplify the implementation of client and server applications by wrapping the complex connection extablishment process. Handlers realize an application protocol abstraction, which can be used to form full-featured network protocol stacks.

At last, ACE provides different services for building distributed software. Components that are commonly used in this field, such as logging-, name-, or time-servers are included. This speeds up the process of designing and implementing such software while making it more reliable by providing a thoroughly tested version.

Noteworthy about the design of ACE is the ability to use explicit dynamic linkage. It enables loading and unloading necessary libraries at runtime. While this approach adds an implementation overhead to the framework, it can reduce the initial load time and the memory requirements of applications. This allows resulting software to run even on very restricted hardware, which is a feature only very few frameworks provide.

Since the abstraction relies on the provided OS features to transport data, only UDP and TCP are supported. This can be a problem for more specialized applications that require more unusual transport protocols. However, because an application protocol abstraction is provided, such protocols can be implemented at the application layer on top of UDP.

Discussion

The implementation of the library is very sophisticated and has proven to be reliable. A very high level of abstraction and versatility is provided by ACE by encapsulating every bit of functionality into components. This approach makes it very easy for programmers to select specific functionality for their use-cases.

The scalability of the framework is also not an issue. With the event-demultiplexing implementations, resulting software can be built in a very scalable and efficient way. Using such a solution drastically limits the management overhead that is added by large numbers of connections. Thus allowing to build very efficient, large scale distributed software systems.

A key feature is the inclusion of an application protocol abstraction, which allows programmers to build reliable and specialized network stacks. Even the limited number of supported protocols can be expanded using this feature, which boosts the versatility of the framework. Thus, making ACE very attractive for implementing network communication in software written in C++.

The field of application of CAF is manifold, which opens a variety of different use-cases. Hence, the new network design should focus on achieving a level of abstraction and composability comparable to the one of ACE. It would enable the framework to be implemented for more specialized use-cases. However, CAF is a framework for actor programming, and not a networking framework. Hence, the variety of included services, such as name- or time-servers are unnecessary and should be omitted.

4.3 An Architecture for Transport Services

The Architecture for Transport Services (TAPS) draft [30, 31, 32] is an effort by the IETF to standardize the many interfaces for transport protocols. Currently, many APIs for transport protocols exist, which makes changing the underlying transport very difficult. Implementing asynchronous transport abstraction layers is a general, standardized task, which does not differ greatly between the implementations. Modules such as a multiplexer and a socket API abstraction are necessities, which do not leave much of a margin for other approaches. Thus, the TAPS working group aims to provide twofold: a standardized, general interface and a general asynchronous transport abstraction architecture for transport protocols defined by the IETF.

Figure 4.4 shows the proposed API model of TAPS. It consists of APIs for resolving and for stream and datagram communication, which form an abstraction of the provided functionality in the Linux kernel.

The draft includes features that provide improvements over the current C socket API. It provides:

- Common APIs for common features.
- Access to specialized features.
- An event-driven asynchronous API.

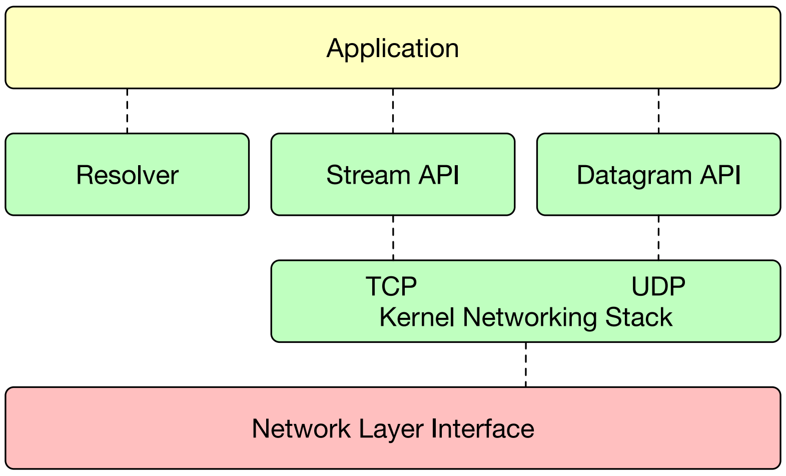


Figure 4.4: Overview of the TAPS API model

- A message-oriented data transfer.

TAPS aims to resolve the problem of differing APIs for common features by unifying them through a common, high-level API. Functions such as `connect`, `receive`, `send` , and so on provide similar functions for the caller. Thus, a common interface for them is proposed that fits for all these functions. However, some applications require a more specialized interface to allow fine-grained access to the underlying protocols. Such functions are exposed independently of the common API to ensure a flexible design.

Nowadays, network communication is often designed to be asynchronous or at least pseudo-asynchronous by using an event-driven model. Thus, the draft relies solely on event-driven I/O and handles every call to the API asynchronously by using callbacks. For example, the `receive` call would not be handled instantly but register a callback that is called after receiving the data. This allows to handle the I/O asynchronously without having to block on system-calls. Furthermore, it removes the multiplexing facilities that are currently still necessary for such an approach.

Three communication models can be used for communicating over the network: datagram, stream, or message-oriented communication. TAPS relies on message-orientation for the representation of received and outgoing data. Messages have the benefit that they contain any number of bytes while being self-contained with a defined size, thus any amount of data can be transmitted. However, native stream-oriented protocols require a translation between the two models since, for example, Hypertext Transfer Protocol (HTTP) uses character delimiters to identify the end of a section. For this, so-called framers are proposed, which have to be situated between the application and the transport layer. These entities read the required amount of bytes and pass it as a single message to the application layer. Since this can be different for every protocol, these framers can be exchanged to fit the requirements of the application.

Discussion

The APIs of transport protocols usually provide an abstraction of very similar functions. Thus, TAPS aims to provide a generic interface for all kernel-provided networking capabilities. This effort to standardize the interfaces for many protocols simplifies exchanging transport protocols in resulting software. Since this unifies the interfaces, the underlying transport protocols can easily be exchanged without the need to alter the code. Furthermore, the inclusion of multiplexing facilities in the abstraction allows relying on a library implementation rather than rewriting the routines every time.

CAFs network abstraction design can benefit from the design proposed by the TAPS working group. The general API proposed for any transport protocol would be a very good addition to the design, since it allows swapping protocols easily. Relying on a message-oriented communication model for communication with the application layer is something that should be considered for the new design. It provides a good compromise between datagram and stream-oriented transport protocols. Arbitrarily sized messages can be processed in the stack without the necessity of streaming or datagram-related overheads.

However, the framing component used in TAPS is not sufficient for the use-case in CAFs networking design. The general idea of using framers that create well-formed messages for the following layers is good but not portable enough. For the redesign in CAF, such components should be reusable for other configurations, which is not possible with application-dependent framers.

4.4 Comparison

The examples of related work are manifold and all of them feature different properties that they provide. Thus, for a complete overview of the criteria Table 4.1 lists all

	CAF	Netty	Boost Asio	ACE	TAPS
Configurable Application Layer	✓	✓	✗	✗	✗
Exchangeable Transport Protocols	✓	✓	✓	✓	✓
Inclusion of Custom Transport Protocols	❍	✗	✗	✗	✗
Communication Model	Message-oriented	Transport Dependent	Transport Dependent	Transport Dependent	Message-oriented
Asynchronous/ Event-driven I/O	✓	✓	✓	✓	✓

Table 4.1: Comparison of the properties of the frameworks shown in Chapter 4. Reference: ✓: true, ✗: false, ❍: partially

discussed frameworks. The criteria that have been shown in the table are aimed at full-featured network stacks. Hence, the criteria also include configurability and application layer protocol abstraction.

Most of the frameworks are aimed solely at providing a thorough transport abstraction. The proposed redesign of the stack in CAF and the approach taken by Netty aim for a broader abstraction though. Hence, both designs allow including application protocols.

The exchangeability of transport protocols is a criterion that all the frameworks fulfill. However, all of them rely on the transport protocols provided by the OS (e.g. TCP, UDP, SCTP, etc.). In CAF, the inclusion of other experimental or custom protocols would be possible but poses a significant implementational overhead.

An important design decision of stacks is the used communication model. For future portability of the code this is very important since the following layers either have to be adapted, or adaption layers have to be included in the design. Of the five discussed frameworks, only two have defined a specific communication model. All others use the model defined by the underlying transport protocol, which could potentially hinder the portability of the resulting stack.

This leads to the conclusion of this chapter. A network stack design should incorporate threefold:

1. Exchangeable transport protocols.
2. An application layer abstraction.
3. A well-defined communication-model.

5 Design of the Network Stack

Network stacks provide an abstraction of the given OS features to allow communication with remote nodes over the network. This is commonly done by splitting the functionality into transport and application layers that encapsulate routines for processing and transmitting data. The transport layer uses different transport protocols, while the application layer is using application layer protocols. A new design for both layers is proposed by this thesis that will be covered, explained, and justified in this chapter.

5.1 Goals of the Redesign

In Chapter 3, issues with the current design have been outlined. These issues combined with new requirements for a network abstraction design can be rephrased in the form of goals for the new network stack design. The goals can be roughly divided into four separate groups: code complexity, composability, scalability, and performance of the network stack.

5.1.1 Code Complexity

Code complexity directly affects future efforts to maintain and extend the network stack. Thus, transparent class hierarchies and evident relations between components are necessary to simplify such efforts. Code complexity can be measured with the help of common software metrics.

Components in the design should have a clear, well-defined function in the stack. This mitigates confusing similarities between components as well as complex hierarchies that hide the actual composition. Additionally, it allows using more concise names for components which helps to promote the purpose they have in the stack. Both of these topics relate to the correctness (e.g. following the specification) of the design.

Furthermore, a clear distinction between programming concepts should be done. Function calls in one direction and messages in the other hide program paths that help to understand the program flow. Thus, the redesign should opt for a single concept whenever feasible to make program paths as obvious as possible.

This leads to three goals:

1. The purpose of each component has to be clear.
2. Functionality should be well-defined.
3. A single programming concept should be used across the stack.

5.1.2 Composability

Composability of the design defines the ability to exchange and assemble components in any order. The included protocol abstraction used by the stack should be designed to be self-contained, to strictly separate the concerns of all instances. This approach decouples the components, which mitigates the need to change the surrounding code when the stack is altered.

Transport protocols are always used independently, without the need to compose different protocols. Hence, the transport protocol abstraction only has to provide a stable API that can be used by other components. The Application protocol abstraction is required to be composable since application protocols are designed to be composed. Thus, the design of the application protocol components should provide the possibility to compose them to form a stack.

This leaves four goals for the composability of the design:

1. Coupling between components should be as loose as possible.
2. Exchanging either transport or application protocol abstractions should be easy.
3. Exchanging components should not require changes in other components.
4. Application protocol abstraction has to be composable.

5.1.3 Reusability

The process of designing components can be time-consuming and complicated. Since the design of this network stack should be composable, components that have been designed for other compositions should be reusable. This leads to a single goal for the new design of the network stack:

1. Components should be reusable for other stack compositions.

5.1.4 Scalability

The scalability of a network stack can be measured by the changes in performance or memory impact with a specific change of load in the system. For example, the network stack is required to handle any number of actors communicating with many remote nodes. This condition should not affect the performance of the stack in any way.

Three goals can be formulated:

1. The stack should scale at most linearly with the number of remote nodes.
2. The number of actors that are involved in remote communication should not have an impact on resources on the remote node.
3. The throughput of the stack should scale at most linearly with the size of the messages that are transmitted.

5.1.5 Performance

Software that communicates over the network relies directly on the latency and the throughput of the underlying network abstraction. High latency and low throughput can impair the performance significantly which would render timing-sensitive applications useless. Thus, the new design should aim to provide an abstraction over the OS API with low latency and high throughput.

The goals for the performance of the design are:

1. Aim to add as little latency to the OS network abstraction as possible.

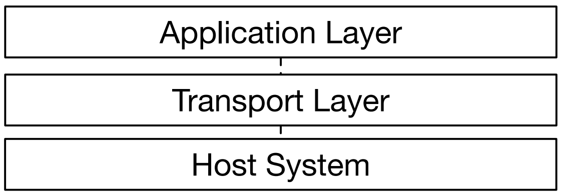


Figure 5.1: Visualization of the layers in the new design

2. The throughput of the resulting stack should be comparable to similar solutions such as the Message Passing Interface (MPI).

5.2 The Network Stack Design

The proposed network layer abstraction in CAF has been redesigned from the bottom up. It includes facilities for I/O multiplexing, as well as including application protocols and handling the transport of data. The functionality of the design is separated into two layers: Transport and application layer. Figure 5.1 shows the order of the two layers between the host system and the actor layer.

5.2.1 Fundamental Design Decisions

The design of the network stack is based on fundamental decisions that have been made beforehand. Before introducing the new design, these decisions are laid out and explained in the following section.

Communication Models

There are three viable communication models that could be used within the network stack: Datagram, Stream, or message-oriented communication.

Within the application layer, the model of choice is message-orientation due to the variable size of messages that are passed through the stack. It should be possible to include any number of applications in the stack, thus a payload can be extended by any number of bytes from each layer. Thus, the communication model within the application layer needs to allow variably sized messages to accommodate the increasing amount of data.

Transport protocols for any of the three communication models exist, which should all be usable in the network stack. However, datagram-oriented protocols can neither handle stream nor message-orientation, which poses a problem. The communication model used within the transport layer depends on the underlying transport protocol that is used.

Scatter-Gather I/O

Processing messages that are continuously expanded would require reallocating memory blocks and copying the contents of the message. Since both of these are expensive tasks, the new design requires a more appropriate solution for this problem. Thus, scatter-gather I/O[1] is the chosen approach for the representation of messages within the stack.

Scatter-gather I/O allows scattering messages across any number of buffers, which can then be written to a socket in a single system call. A common approach is to issue many write-calls to write all the buffers sequentially to the socket, which is expensive since `write` is a system call. Another approach is to concatenate the buffers before issuing the write call, which adds a copy-overhead to every `write`.

However, the API for sending such messages introduces templates, which would not be necessary without this approach. Furthermore, a possible slicing layer for packet-oriented transport protocols will be more complicated to design because a single message is scattered across multiple buffers. Thus, this approach also has the caveat of complexity that is introduced to the design.

5.2.2 Transport Layer Abstraction

The transport abstraction layer handles the receiving and transmitting of data by providing a protocol-agnostic interface for accessing the socket API. The new design proposes an abstraction that is based on four main components: The multiplexer, the endpoint manager, the transport interface, and the transport extensions, which are optional. Figure 5.2 visualizes the composition of the three components on the transport layer.

[1]`https://www.gnu.org/software/libc/manual/html_node/Scatter_002dGather.html`

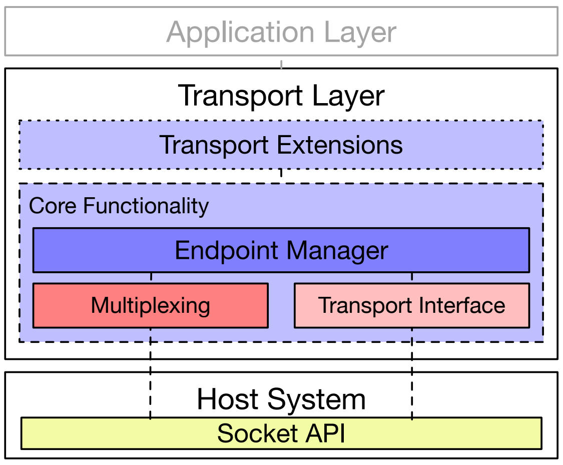


Figure 5.2: Visualization of the Transport Layer in the new design

Multiplexer

The multiplexer is responsible for two tasks in the design: multiplexing socket events and scheduling event handlers. It relies on a reactor (for example `epoll`, `poll`, or `select`) that monitors sockets in its pollset for the multiplexing. For the scheduling of event handlers, the multiplexer waits for an event to occur on any of the sockets, which it then passes on to the handler. The multiplexer is executed by a dedicated thread, thus it is the only active component in the whole design.

Noteworthy about the multiplexer is that in the current design there is only one instance. This limits the stack to be strictly single-threaded, which allows omitting to synchronize the access to components within the stack.

The use of the multiplexer is critical for the design because it enables serving numerous sockets concurrently. Without it, either a thread has to be started per socket or the number of sockets has to be limited. Both of these approaches would drastically limit the scaling capabilities of the whole design.

Endpoint Manager

The endpoint manager is a component that is situated above the multiplexer in the network stack. First, it functions as local representation for remote endpoints by assigning a single socket to the instance. This is possible when using TCP because the socket represents exactly one remote endpoint. For connectionless protocols such as UDP, this abstraction is not applicable because a single socket can be used for communicating with multiple remote nodes. Thus, a dispatching solution has been included in the design that is explained in Section 6.3.

Second, it functions as an event-handling entity that is scheduled by the multiplexer for reading and writing. The instance is added to the pollset of the multiplexer, which maps handlers to their corresponding sockets and triggers them according to the occurring socket-events.

Third, the endpoint manager is responsible for storing outbound messages until they can be written to the socket. When an actor sends a message to a remote node, the message is enqueued in the endpoint manager. The manager stores the messages in a queue until the multiplexer schedules it for writing. This approach decouples the two threads of execution (actor and multiplexer) and allows the communication to be asynchronous.

The access to the socket layer is provided by dedicated transport interface components. These are used by the endpoint managers to access the socket layer during a read or write event. By encapsulating all protocol-specific functionality in such components, exchanging transport protocols is simply done by exchanging them. Thus, enabling to reuse the endpoint manager for a variety of different protocols without the need to alter it.

Transport Interface

The transport interface provides a consistent, general API for reading and writing data from and to sockets. Since routines for receiving and transmitting data can vary greatly between different transport protocols, such an abstraction is required. The proposed design provides a `datagram_transport` and a `stream_transport` for UDP and TCP.

The component has the task of reading and writing data from and to the socket when it is called by the endpoint manager. For reading, the instance owns a read buffer, in which the received data is stored before passing it to the application layer. For writing, the instance dequeues a message from the endpoint manager, passes it through the application stack and writes the processed data to its socket. Both actions require an application instance, which can be used for the processing of data, thus an instance of the application stack is included in the transport abstraction.

```cpp
/// Stream oriented transport interface.
template <class NextLayer>
class transport {
  /// Called by the endpoint manager when the transport can
  /// read data from its socket.
  bool handle_read_event(endpoint_manager&);

  /// Called by the endpoint manager when the transport can
  /// write data to its socket.
  bool handle_write_event(endpoint_manager&);

  /// Queues the message scattered across multiple buffers
  void send_data(id_type, span<byte_buffer*>);

  /// Returns the next cached header buffer
  byte_buffer next_header_buffer();

  /// Returns the next cached payload buffer
  byte_buffer next_payload_buffer();

private:
  /// Buffer for data that was read from the socket.
  byte_buffer read_buffer_;

  /// The next layer of the stack.
  NextLayer next_layer_;
};
```

Listing 5.1: Proposed API for the transport protocol abstraction

Fig. 5.1 shows the proposed interface of the transport interface. It provides two callbacks for I/O events from the multiplexer (`handle_read_event`, `handle_write_event`). `handle_read_event` will read data from the socket and pass it to the application layer for processing and delivering actor messages. `handle_write_event` writes the data that was received by the application layer after the processing of an outgoing message. The application layer calls `send_data` with the processed message or packet scattered across any number of buffers. For datagram-oriented protocols a slicing layer has to be included which slices the message into feasibly sized packets.

The other two functions that are provided are used for buffer recycling on the application layer. Each time a buffer is required, `next_header_buffer` or `next_payload_buffer` can be called to obtain a cached instance. Headers are usually far smaller than the payload, which led to the decision to separate buffers into these two types. The performance impact of allocating memory for buffers, which can be quite a costly task, can be mitigated by this. After the processed data has been written to the socket, the used buffers are cached again for future use.

5.2.3 Transport Extensions

Transport protocols can be extended by transport specific protocols such as Transport Layer Security (TLS), QUIC, or HTTP. Such protocols are designed as application protocols but the functionality they provide is coupled to the transport protocols underneath. QUIC for example, is a transport protocol that is designed to extend UDP. Thus, it has to be situated in the transport layer and not on the application layer, since it extends the provided functionality of UDP. For this, the new design proposes an obligatory transport extension layer, which can be used to extend existing transport protocols.

The transport extension is transport protocol specific, which implies coupling the communication model of the components to the underlying transport protocol. For this, the extension component can provide either of three handle functions: `handle_data` for stream-oriented transport protocols, `handle_message` for message-oriented transport protocols, and `handle_datagram` for datagram-oriented protocols. Furthermore, to add type-safety to the components, a tag-type for the input and output model is introduced, which can be checked at compile time.

The proposed API of the transport extension component is shown in Listing 5.2. In the beginning the tag types that define the input and output models are defined and checked against the tags of the next layer. Since in this example a stream is expected as input type, a `handle_data` function is included.

5.3 Application Layer Abstraction

With the redesign of the network stack, a new design for including application protocols is proposed. It allows including protocols in the form of encapsulated components that can be stacked in a recursive approach. This enables composing different network stacks in a very versatile way. Figure 5.3 visualizes the components on the application layer.

```
template <class NextLayer>
class transport_extension {
  static constexpr type input_type = type::stream;
  static constexpr type output_type = type::message;
  static_assert(NextLayer::input_type == output_type);

  /// Called by the previous layer, with the data that should be
  /// processed.
  template <class Parent>
  void handle_data(Parent&, byte_buffer&);

  /// Called, when a message should be sent.
  template <class Parent>
  void write_message(Parent&, actor_message&);

  /// Called by the next layer to send messages.
  template <class Parent, class... Ts>
  void send_data(Parent&, Ts&... buffers);

private:
  /// Next layer of the stack.
  NextLayer next_layer_;
};
```

Listing 5.2: Proposed API for the transport extension components

5.3.1 Application Protocol Abstraction

Protocols only have to know the following instance, which ensures weak coupling between the individual instances. The weak coupling allows to stack an arbitrary number of different protocols. Furthermore, by stacking the components instead of using a monolithic approach, designing and configuring such stacks is very simple. Each application protocol can be designed individually, which eases the design process and allows to reuse it.

Each protocol has to provide a general interface shown in Listing 5.3. It requires two functions: `handle_message` and `write_message`. `handle_message` processes messages that have been received. Additionally, the name of the function reflects that the processing is message-oriented, which makes the usage more clear. `write_message` processes outgoing messages. Both functions pass the results on to the next layer until the last application is reached. If actor communication is required, the last application has to be BASP, which is used in all examples of this thesis.

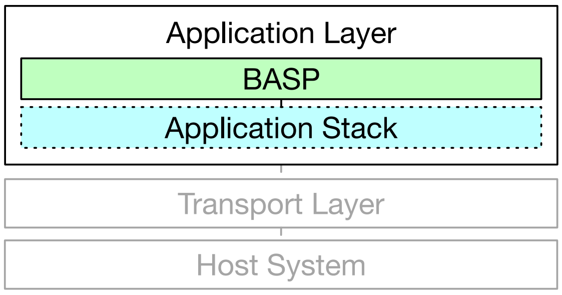


Figure 5.3: Visualization of the application layer in the new design

```
template <class Application>
class application {
  /// Called by the previous layer, with the data that should be
  /// processed.
  template <class Parent>
  void handle_message(Parent&, byte_buffer&);

  /// Called by the transport, when a message should be sent.
  template <class Parent>
  void write_message(Parent&, actor_message&);

  /// Called by the next application to send messages.
  template <class Parent, class... Ts>
  void send_message(Parent&, Ts&... buffers);

private:
  /// Next layer of the application stack.
  Application next_layer_;
};
```

Listing 5.3: Basic API of the application protocol abstraction

Stacking the application instances is done in a recursive pattern. Applications need a template argument `template <class Application>` to be composable with other instances. Each application instance then defines a member of the template type for the next layers instance. The last application in the stack does not have this template to break the recursion. Thus, for building a network stack, two types of applications are necessary: Intermediate application protocols and top-layer application protocols (e.g. BASP).

5.3.2 BASP

The design of BASP needed reconsideration as well, since the previous iteration of it included more functionality than necessary. Thus, the redesign proposes a slimmed-down version of the protocol.

BASP is now included in the stack in the form of application protocols, which are added to the protocol stack as the top-layer. With the application-based approach, multiple BASP instances can be instantiated. This simplifies the process of receiving and processing data, since each BASP instance only has to handle a single endpoint.

The protocol can now be transport agnostic, which improves the composability of the stack. Previously the protocol had to distinguish between the underlying transport protocols to handle the data the right way. This is now obsolete, which eases the process of including other transport protocols.

Furthermore, the overlay networking functionality discussed in Section 3.5 has been removed from the stack. This allowed to strip the routing tables and handling routines from the protocol, which slims down the design even further.

5.3.3 Universal Resource Identifier

The new design proposes the addition of an Uniform Resource Identifier (URI) [33] for the use within the network stack. URIs allow to unambiguously identify resources across the network. Hence, they can be used to simplify the process of resolving actors remotely

```
caf:udp://actor-framework.org:1337/name/source
\_____/   \______________________/ \_________/
   |                 |                  |
 scheme          authority            path
```

Listing 5.4: Example of an URI

A basic URI consists of a scheme, an authority, and a locator (path) as shown in Listing 5.4. The scheme can be used to imply a specific transport protocol such as TCP or SCTP in addition to the application stack composition on top of that. The example uses the scheme `caf:udp`, which would imply the use of UDP as transport, with ordering,

```
class middleman {
  /// Publishes the given actor `whom` on `path`
  void publish(actor_handle& whom, const std::string& path);

  /// Unpublishes the actor on `path`
  void unpublish(const std::string& path);

  /// Tries to resolve a remote actor on `locator`.
  /// If no connection exists, the call will establish a
  /// connection to the remote node first.
  expected<actor_handle> remote_actor(const uri& locator);

private:
  /// Stores the global socket I/O multiplexer.
  multiplexer_ptr mpx_;

  /// Stores all available backends for managing peers.
  middleman_backend_list backends_;
};
```

Listing 5.5: Proposed API of the middleman

delivery, and BASP. An authority holds addressing information of the host, such as an IP or a domain and a port. Locators specify the path on the remote system.

5.3.4 User Facing API

The new design also requires a user facing API to grant access to the functionality of the stack. Thus, a so-called `middleman` is proposed, which provides an API to do so.

Listing 5.5 shows the proposed interface of the component. The provided functionality is limited to two functions: `publish` and `remote_actor`. `publish` publishes a local actor for the given path so that it can be resolved remotely for actor communication over the network. `remote_actor` does twofold: connecting to the remote node if necessary and resolving remote actors. If a connection to the remote node already exists, the connecting step is omitted and the existing connection is used.

Resolving an actor remotely is a core functionality of the network stack and important for building distributed systems. Figure 5.4 visualizes the resolving of `caf:tcp://actor-framework.org:1337/name/source`. Five steps are necessary for this:

1. Acquire the corresponding backend for the scheme.

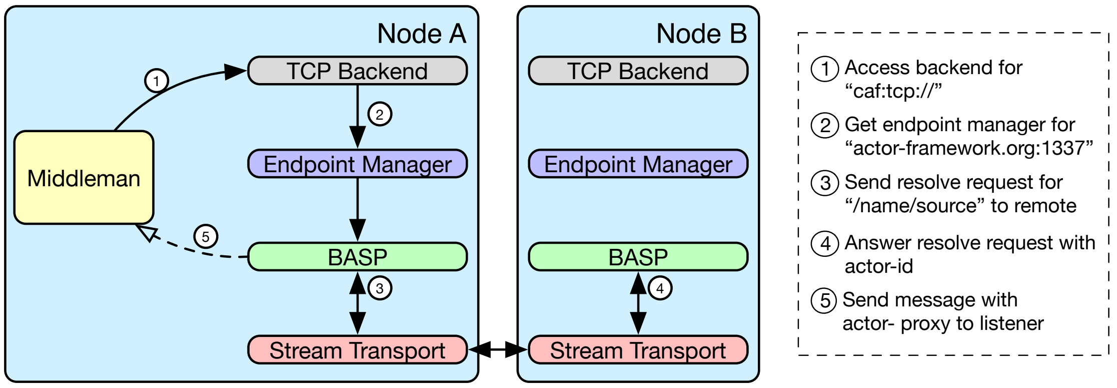

Node A
TCP Backend
Endpoint Manager
BASP
Stream Transport
Middleman
1
2
3
5
Node B
TCP Backend
Endpoint Manager
BASP
Stream Transport
4
1 Access backend for "caf:tcp://"
2 Get endpoint manager for "actor-framework.org:1337"
3 Send resolve request for "/name/source" to remote
4 Answer resolve request with actor-id
5 Send message with actor- proxy to listener

Figure 5.4: Example of resolving an actor using an URI

2. Obtain the corresponding endpoint manager for authority.
3. Send a resolve request with the path to the remote node.
4. Remote node sends resolve response with actor id
5. Create actor-proxy and send handle to the middleman.

Using URIs for identifying actors simplifies the design process. Actors can now be referenced by name, which makes the code more expressive and the purpose of specific actors more clear. Furthermore, the new process allows including precise error reporting which goes beyond „Could not connect to node".

The middleman functions as top-level entity of the whole stack. `middleman_backends` are used to represent specific network stack compositions, which the middleman can hold an arbitrary number of. Each composition consists of a specific transport protocol and an application stack that is composed of any number of application protocols. Figure 5.5 visualizes this approach.

In the figure, a configuration with two backends is shown: The CAF TCP backend and the CAF UDP backend which uses ordering and delivery options. In both configurations, BASP is used as top-level application because the stacks are designed for actor communication. Noteworthy about this visualization is the difference between the stack layouts for UDP and TCP. TCP can use the individual sockets for representing the remote endpoints, hence the backend will simply store multiple `endpoint_managers` . In this case, each endpoint manager represents a single remote endpoint and stores their specific application stack instance. For UDP this is not possible which requires the

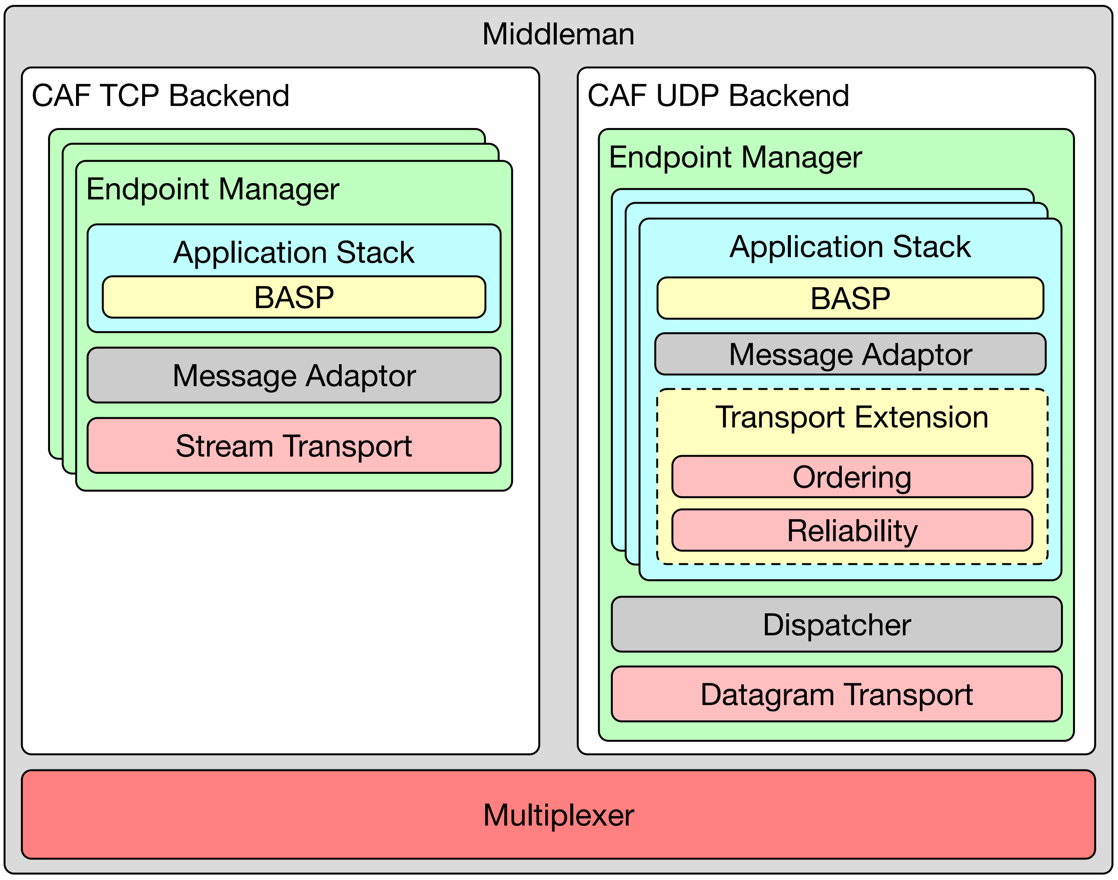


Figure 5.5: Visualization of the high-level design

inclusion of a dispatching solution in the stack. The CAF UDP backend holds only a single endpoint manager which uses the dispatcher to multiplex between any number of application stack instances.

5.3.5 Serializing

As discussed in Sec. 3.2.1, serializing is a costly task that has to be done at a fitting place, to limit the introduced performance impact. Additionally, the process can be done concurrently which can mitigate the runtime overhead significantly. This leaves three different solutions of where to serialize: In the actor proxy, the BASP application, or concurrently.

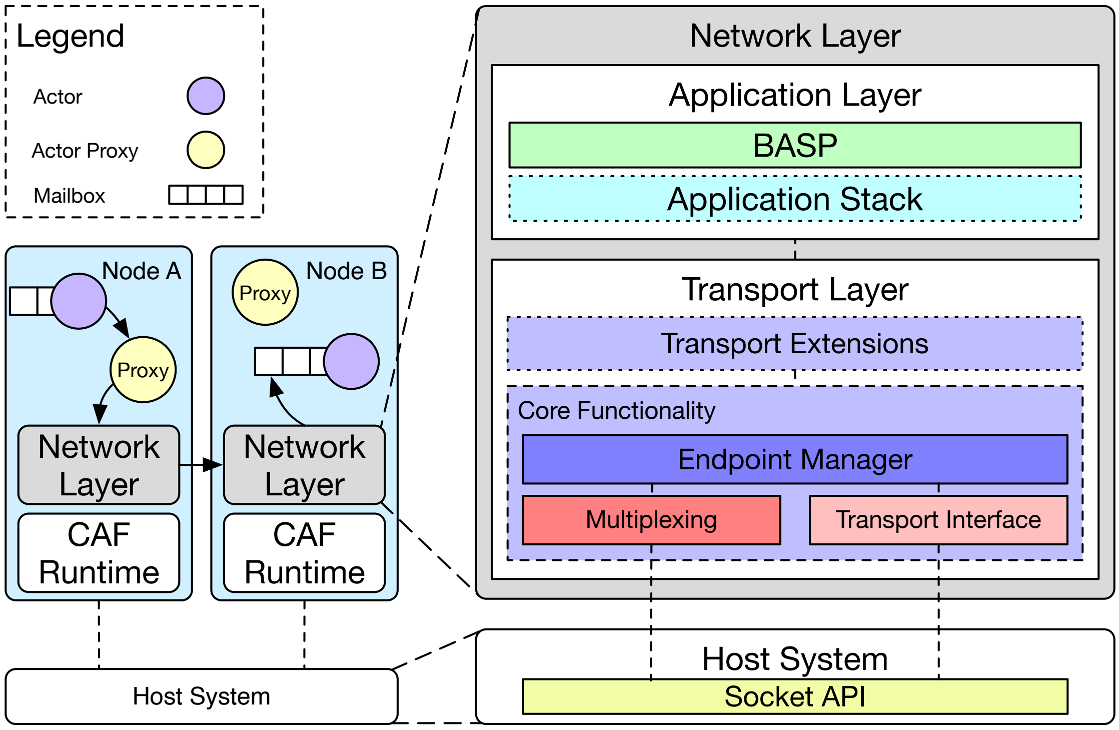


Figure 5.6: Overview of the proposed design and its components

Serializing in the actor proxy would imply serializing the message before enqueueing it to the endpoint manager. The overhead is thus added to the runtime of the sending actor proxy. This approach is beneficial when lots of actors communicate with remote nodes because it takes the performance overhead out of the multiplexing-thread. Moving this task to the BASP application does the opposite: It moves the overhead from the actor-proxy to the multiplexer. This can be beneficial when fewer actors send lots of data to remote nodes.

The last approach is to completely separate the serializing overhead by parallelizing it. This approach does not add any performance overhead to either the actor-proxy nor the multiplexer but will require ordering messages after serializing them. Messaging in CAF is strictly ordered, which can introduce another performance overhead to the process.

Since serializing messages on the application layer allows for buffer recycling, this is the approach that has been proposed.

5.3.6 Actor Communication in the new Design

Sending actor messages using the new design is a straight-forward task. Figure 5.6 visualizes the most important components on each layer for the communication between two actors running on different nodes.

In this example, the actor on node A (actor A) wants to send a message to the actor on node B (actor B). To do so, actor A sends a message with the content to an actor proxy that represents the remote actor B on the local node. The proxy enqueues the content of this message to the endpoint manager, which is then registered for writing in the multiplexer. When the write-event is triggered, the endpoint manager will dequeue the message and pass it to the application stack for processing.

In the stack, the top-level application (in this case BASP) has to serialize the message. After serializing the message, the resulting bytes are passed to the next application. The last application will pass the message to the transport, which will write the bytes to its socket.

The multiplexer on node B will trigger a read-event when the data arrives and execute the corresponding endpoint manager. It then triggers the endpoint manager to read the data from its socket and pass it to the application stack. As soon as the data reaches the BASP application, the actor message is reassembled and delivered to the receiving actor.

5.4 Example Configurations

The new design allows composing any number of protocols to form full-featured network stacks. To give some perspective about the capabilities of the design, the following section introduces three exemplary compositions of stacks.

Figure 5.7 shows three exemplary compositions of the network stack design.

Figure 5.7a shows a composition that uses BASP on the application layer as top-level application. On the transport layer, it uses HTTP and QUIC as transport extensions, which are using UDP as a transport layer. This composition allows to tunnel actor communication through HTTP on port 80 or 443, which can mitigate the interference of firewalls.

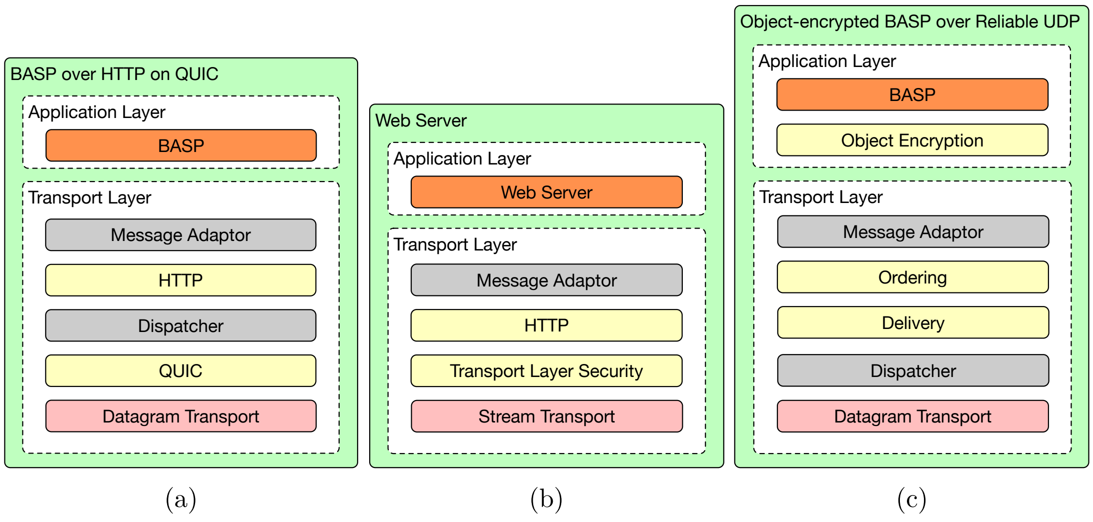


Figure 5.7: Examplary network stack compositions

Figure 5.7b shows a composition, with which CAF could be used as a web-server. For this, the transport layer is composed of HTTP on TLS for secure transmission, which is transmitted over TCP. This composition allows to build an actor-based web-server that could easily handle vast amounts of clients, which are served by actors.

Figure 5.7c shows a composition that uses the default UDP transport of the new design in addition to an object-encryption application. This additional application would be situated on the application layer and encrypt every message, independently from transport security. Thus, this composition allows secure communication over a reliable UDP transport.

5.5 Discussion of the new Design

The new design of the network layer is significantly simpler than the current design. Transport protocols can easily be exchanged with the proposed transport layer design. Furthermore, the new design of the transport abstraction makes it very simple to include any other transport protocol to the stack. The proposed application design enables the composition of full-featured protocol stacks. These can easily be configured by composing protocol components that could be reused for any stack layout.

By including the middleman backend, each stack composition can be included in parallel to other compositions. This allows to use the network stack for multiple fields of application at once.

In the following section the proposed changes are discussed according to the goals that have been laid out in Section 5.1.

5.5.1 Code Complexity

The new network stack features explicit and transparent hierarchies across the design, which makes relations between the components much more clear. Components are now separated across distinct layers, which allows to easily distinguish their function and purpose in the stack. Furthermore, there are no complex hierarchies among the components of the stack, which mitigates problems that arise with non-evident relations between them. The clear separation of concerns removes implicit dependencies among entities, which further simplifies both comprehensibility and limits the code complexity.

However, the lowering of complexity by including a clearly separated layering approach for the stack also reintroduced complexity. Since composing the stack relies solely on templates, many of the proposed components will require at least one template argument. Furthermore, keeping the interfaces of the components as clean as possible required further addition of templated functions across the stack. However, the templates that included to the design are intentionally kept as simple as possible.

5.5.2 Composability

The proposed design increases the composability of the stack with the introduction of clearly confined components. Both transport and application layers have been decoupled by this design change, which allows tailoring either of them to the needs of the field of application. The application layer can now include any number of protocols, which can be designed separately from each other. The stacking approach in this design does not rely on strong coupling between the protocols, but a general API for passing data between the layers.

Furthermore, transport protocols are now strictly decoupled from the application layer, which enables easily exchanging them for any other transport protocol. This improves the composability of the stack significantly, while still offering a high level of abstraction of the lower levels.

5.5.3 Scalability

BASP is now included in the design in the form of an application protocol that can be added on the top-layer of the stack. Additionally, application stack instances to a specific endpoint, which removes the necessity of the monolithic `broker`-based approach. Hence, this approach should scale at most linearly in terms of memory and improve the performance of the stack. Furthermore, the unnecessary extra layer, added by the design with `brokers` has been removed completely, which essentially halves the number of necessary write-events per outgoing message. Both claims about the scalability will be evaluated in-depth in Chapter 7.

6 Implementation of the Network Stack

In this chapter the implementation of the new design for the network stack is shown. Some noteworthy implementation specifics are explained and the reasoning behind the implementation decisions is discussed.

6.1 Transport Abstraction

The transport abstraction classes are key components in the design, which provide read and write access for the socket layer. The implementation currently includes two transport protocols: UDP and TCP. Other protocols, such as QUIC are planned to be implemented and included in the future.

The general process of writing a message is divided into three distinct steps. First, the message that should be sent is enqueued in the transport abstraction after being processed in the application layer. The second step is writing the data to the socket and the third step is recycling the buffers that contained the message.

Enqueuing is done by calling the `send` function with the message scattered across any number of buffers. The buffers are then stored in a write queue until they can be written to the socket. Afterwards, the written buffers are „recycled“, which is done by separating the buffers into header or payload buffers and storing them in the according caches.

6.1.1 Enqueuing Data to the Transport

For TCP the routines are implemented following a stream-oriented communication model. Thus, the `stream_transport` holds a write queue with the type `std::deque<std::pair<bool, byte_buffer>>`, which allows to enqueue and dequeue buffers sequentially. `std::deque` was chosen because accessing it from the front or the back is fast, which is exactly what is needed for a stream-oriented transmission of data.

```
/// Send function for stream-oriented transport protocols
void send(id_type, span<byte_buffer*> buffers) {
  if (write_queue_.empty())
    manager().register_writing();
  auto i = buffers.begin();
  write_queue_.emplace_back(true, std::move(*(*i++)));
  while (i != buffers.end())
    write_queue_.emplace_back(false, std::move(*(*i++)));
}
```

Listing 6.1: Send function implementation in the `stream_transport`

Listing 6.1 shows the implementation for the stream-oriented `send` function. First, the endpoint manager is registered for writing, if it is not currently registered. This step ensures that the multiplexer executes this transport as soon as a write-event is triggered. After that, the buffers are enqueued in the write queue together with a flag that signals whether the buffer is a header or a payload buffer. Storing the buffers this way allows accessing the buffers sequentially without losing the header or payload buffer classification.

For UDP this process is datagram-oriented, thus a different approach had to be implemented. Datagrams have to be sent as a whole, which requires to bundle the buffers of a single packet together so that they can be sent in one piece. For storing datagrams, the datagram transport owns a packet queue, which is of the type `std::deque<packet>`.

The `packet` struct is an abstraction that bundles all buffers that belong to a datagram. It also eases the storing in the packet queue of the datagram transport, which is a very simple task due to the abstraction. Datagrams are passed in scattered representation as well but should have a maximum accumulated size of the Maximum Transmission Unit (MTU). Listing 6.2 shows the implementation of the `send` function in the datagram transport.

```
void send(id_type id, span<byte_buffer*> buffers) {
  if (packet_queue_.empty())
    manager().register_writing();
  packet_queue_.emplace_back(id, buffers);
}
```

Listing 6.2: Implementation of the send function in the `datagram_transport`

Again, the endpoint manager is registered for writing, so that the multiplexer can trigger a write-event when the socket is ready to take more bytes. After that, the buffers are stored in the form of a packet instance in the packet queue.

6.1.2 Writing Data to the Socket

Routines for writing data to sockets are different for every transport protocol. The stream transport abstraction writes bytes sequentially and without preserving message borders, while datagram-oriented transports preserve the bounds by sending datagrams with a fixed size.

```
while (!write_queue_.empty()) {
  auto& buf = write_queue_.front().second;
  auto data = buf.data() + written_;
  auto len = buf.size() - written_;
  auto write_ret = write(handle(), make_span(data, len));
  if (auto num_bytes = get_if<size_t>(&write_ret)) {
    written_ += *num_bytes;
    if (written_ >= buf.size()) {
      recycle();
      written_ = 0;
    }
  } else {
    // handle error
  }
}
```

Listing 6.3: Implementation of the write routine for stream-oriented transport protocols

The write-routine of the stream transport is shown in Listing 6.3. This routine will write as much data as possible from the write queue to the socket in a single call. It will either stop when the write queue has been drained or a socket error is issued which has to be handled. Some errors are unrecoverable, thus this type is propagated to the endpoint manager which will be stopped and removed from the multiplexer. EAGAIN errors signal a full socket buffer, which is handled by stopping the writing process and trying again later.

After the write call is done, the write-result is interpreted and compared with the total amount of bytes in the `byte_buffer`. If all those bytes have been written, the buffer is recycled and moved back to the corresponding cache for future messages.

UDP is implemented with the use of gather-scatter I/O, which allows assembling packets that should be sent from any number of buffers. This routine is shown in Listing 6.4.

```
while (!packet_queue_.empty()) {
  auto& packet = packet_queue_.front();
  auto ptrs = packet.get_buffer_ptrs();
  auto write_ret = write(handle_, ptrs, packet.id);
  if (auto num_bytes = get_if<size_t>(&write_ret))
    recycle();
  else
    // handle error
}
```

Listing 6.4: Implementation of the write-routine for datagram-oriented transport protocols

Since the packet queue holds packet instances, the first step of writing a datagram is to obtain the buffers from the packet. These are then sent in a single write call, which preserves the composition of individual packets. The `packet.id` field holds the endpoint information to which the packet should be sent.

6.2 Packet Writer Decorator

Composing application protocols in the new design is done by recursively stacking them into each other. For this, each protocol component has a `template <class Application>`, with which the type of the following layer is defined. This helps to implement each component without coupling it to another.

Two problems arise from introducing the stacking approach: propagating function calls throughout the stack and appending headers to messages on each layer. Propagating function calls throughout the stack is simple but clutters the individual implementations of the components since some functions are never used by the component itself. Thus, a so-called `packet_writer_decorator` has been implemented, which decorates the calling layer with the transport instance. Functions that always have to be redirected to the transport can be called directly instead of going through every single application of the stack. This approach allows omitting functions in the application implementation, which keeps the components cleaner and simpler.

When messages are processed, the orientation is always clear: Header first, then the payload. Thus, the process of prepending headers to messages on every layer has to be implemented in a general way, since an arbitrary number of layers can be added to the stack. For this, a templated `send` function has been included in the decorator that allows passing any number of buffers to the previous layer. The function is shown in Listing 6.5.

```cpp
/// Send a message consisting of multiple buffers.
template <class... Ts>
void send(Ts&... buffers) {
  object_.send(parent_, buffers...);
}
```

Listing 6.5: `send` function implementation in the `packet_writer_decorator`

The transport protocol abstraction expects a `span<byte_buffer*>`, so that the buffers can easily be moved into the write queue. This step requires a translation of the variadic template, which is done in the worker. The worker provides a `send` function that takes the variadic template and rearranges it into a `span<byte_buffer*>`, which can then be passed to the transport. Listing 6.6 shows the implementation of this function.

```cpp
/// Sends a message consisting of multiple buffers.
/// The vararg `buffers` is rearranged and passed
/// to the transport in the form of span<byte_buffer*>.
template <class Parent, class... Ts>
void send(Parent& parent, Ts&... buffers) {
  byte_buffer* bufs[] = {&buffers...};
  parent.send(id_, make_span(bufs, sizeof...(Ts)));
}
```

Listing 6.6: `send` function implementation in the `transport_worker`

This implementation allows sending any number of buffers, without having to limit or specify the number beforehand. However, it also is a more complex approach to solving this problem, since variadic templates are not easily understood without experience in `C++`.

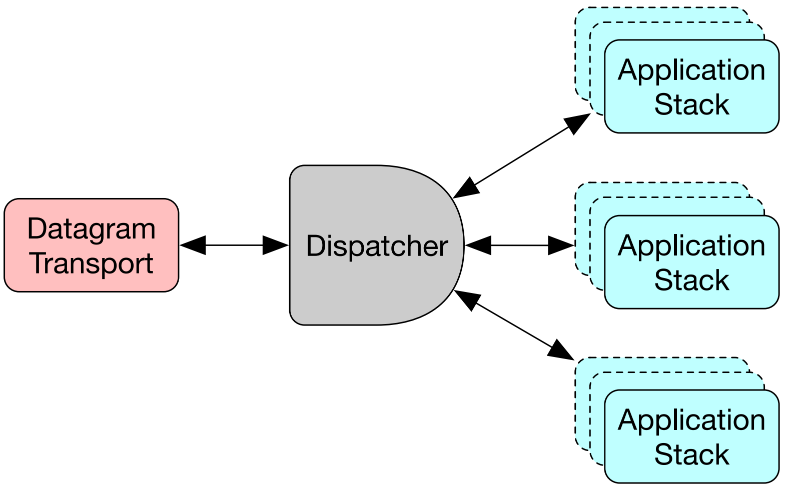


Figure 6.1: Dispatching design for UDP endpoints

6.3 Endpoint Dispatching

The new design revolves around the decision to couple application stack instances to remote endpoints by assigning them to the endpoint managers. This approach is sufficient for TCP because each socket (and thus endpoint manager) represents a single endpoint. Stateless protocols such as UDP on the other hand are not limited to this coupling. Thus, a dispatching layer for such protocols has been included that allows assigning multiple application stacks to a single endpoint manager.

The dispatching layer multiplexes between a single endpoint manager and the corresponding application stack for a remote endpoint. For the dispatching of the received data, the endpoint information is used in the form of an `id_type`. This allows to couple application stacks to remote endpoints for stateless protocols such as UDP too. Figure 6.1 visualizes the approach.

```
if (auto worker = find_worker(id))
  return worker->handle_data(parent, data);
auto locator = *make_uri(to_string(id));
if (auto worker = add_new_worker(parent, make_node_id(locator), id))
  return (*worker)->handle_data(parent, data);
else
  // error
```

Listing 6.7: Dispatching a function call to the corresponding application stack

For the multiplexing, the dispatcher owns two maps in which the stacks are coupled either to the corresponding endpoint information or the URI of the remote node. Mapping the application stacks is done by introducing a component called `transport_worker`, which represents the application stack within the dispatcher. The worker introduces a defined first layer to the stack instances and instantiates decorators to pass to the stack on a function call. Furthermore, it holds the endpoint information, with which the stack is identified. This is necessary for when an application protocol wants to set a timeout, since when the timeout is triggered, the correct application stack has to be identified. Thus, the `set_timeout` function in the `transport_worker` passes the `id_type` to the transport, so that on the way back the timeout can be dispatched according to it.

Passing data to a stack does either of two things: acquiring an existing stack instance or creating a new one. Acquiring an existing instance is done by calling `find_worker`, which searches the map for the corresponding worker and returns it. If the endpoint is unknown by now, a new stack is instantiated and added to the maps before dispatching the data to it. Listing 6.7 shows this routine for the `receive_message` function, which is called after receiving data from the socket.

6.4 Timeouts

Timeouts are essential means to create time-triggered behavior. Protocols can use them to issue retransmits of packets, deliver messages, or simply schedule tasks that should not depend on events that are triggered unreliably. Hence, the implementation of the new network stack provides an API for setting and resetting timeouts.

CAF provides the `actor_clock` for managing and triggering timeouts. This component sends actor messages to actors when a timeout occurred, hence for the implementation of timeouts in the network stack a `timeout_proxy` was included. The proxy receives the `timeout_message` and enqueues it in the endpoint manager queue, which will propagate it when a write-event is triggered. This is necessary, because the `actor_clock` and the network stack are both scheduled in different threads, which could introduce race-conditions without synchronizing the dispatching.

In Listing 6.8 the implementation of the enqueue function in the `timeout_proxy` is shown. It receives the message sent by the `actor_clock`, unwraps the containing timeout_message, and enqueues the values in it to the endpoint manager queue. During this process, the corresponding endpoint manager is registered for writing, so that the enqueued event can be handled by it without relying on an outgoing message.

```
void enqueue(mailbox_element_ptr msg, execution_unit*) {
  if (!dst_)
    return;
  if (msg->content().match_elements<timeout_msg>()) {
    auto tout = msg->content().get_as<timeout_msg>(0);
    dst_->enqueue_event(tout.type, tout.timeout_id);
  } else {
    CAF_LOG_ERROR("timeout_proxy received wrong message");
  }
}
```

Listing 6.8: Implementation of the `enqueue` function in the `timeout_proxy`

Timeout messages contain a type and an id that can be used by the application stack to identify, which timeout has been triggered. The type of the timeout message is used to identify the receiving layer, while the id identifies the specific timeout that has been set before-hand.

Setting timeouts is done by calling the `set_timeout` function on the endpoint manager. The implementation of the function is shown in Listing 6.9.

```
  template <class... Ts>
  uint64_t set_timeout(time_point tp, string tag, Ts&&... xs) {
    auto act = actor_cast<abstract_actor*>(timeout_proxy_);
    sys_.clock().set_multi_timeout(tp, act, std::move(tag),
        next_timeout_id_);
    transport_.set_timeout(next_timeout_id_, std::forward<Ts>(xs)
        ...);
    return next_timeout_id_++;
  }
```

Listing 6.9: Implementation of the `set_timeout` function in the `endpoint_manager`

6.5 Transport Extensions

CAF opts for ordered and reliable communication since it is easier to design and implement software that communicates over the network with these guarantees. Thus, two reliability options for the use with UDP have been implemented so that actor communication can be done reliably even over UDP. Furthermore, sinc IP fragementation is a feature that can introduces higher loss-rates and processing overhead, a slicing layer has been implemented.

Delivery

The reliability option delivery handles retransmitting messages in case they have been lost during the transmission. Each datagram that is sent to a remote node has to be acknowledged, which signals the successful transmission. If such an acknowledgment (ACK) is missing for a certain amount of time, the packet is retransmitted.

For this, the implementation of this reliability option holds a queue for all transmitted packets. Each time a message is passed through this layer, it is copied into a buffer, which is then stored in the queue until an acknowledged is received.

```
if (hdr.is_ack) {
  remove_unacked(parent, hdr.id);
} else {
  // Send ack.
  auto buf = parent.next_header_buffer();
  binary_serializer sink(parent.system(), buf);
  if (auto err = sink(delivery_header{hdr.id, true}))
    // error
  parent.write_packet(buf);
  // Pass remaining data to next layer.
}
```

Listing 6.10: Implementation of the `receive` function of the delivery application

Listing 6.10 shows the implementation of the `receive_datagram` function. Each message can either be an ACK of a previous message or a message from a remote node. If it is an ACK, the message queue is searched for the id of the ACK, which will then be removed. In case of a new message from a remote node, an ACK is directly enqueued to the transport and the message is passed on to the next layer.

```
auto hdr = parent.next_header_buffer();
binary_serializer sink(parent.system(), hdr);
if (auto err = sink(delivery_header{id_write_, false}))
  // error
add_unacked(parent, id_write_++, hdr, buffers...);
parent.write_packet(hdr, buffers...);
```

Listing 6.11: Implementation of the `write_datagram` function of the delivery application

Listing 6.11 shows the implementation of the `write_datagram` function. It processes outgoing messages by copying them into the message queue for eventual retransmission. This is a complex task since the message is scattered across several buffers and also in the form of variadic templates, which cannot easily be concatenated. Thus, an `insert` function has been implemented that takes a variadic template and concatenates it in a single buffer for storing it. The function is shown in Listing 6.12.

The function first asserts that the type of the template is equal to `byte` so that any bugs can easily be found. After that, the buffers from the template are concatenated in the buffer with the help of fold expressions. This construct allows applying a specific action to all template arguments, which is very helpful when using variadic templates.

```
// Inserts variadic template 'bufs' into a single buffer 'buf'.
// Necessary for saving unacked packets until they are ACKed.
template <class... Ts>
void insert(byte_buffer& buf, Ts&... bufs) {
  static_assert((std::is_same_v<byte, typename Ts::value_type> &&
      ...));
  (buf.insert(buf.end(), bufs.begin(), bufs.end()), ...);
}
```

Listing 6.12: Implementation of the `insert` function of the delivery application

Ordering

The reliability option ordering guarantees ordered communication. Every message that is passed through this layer is extended with a sequence number, which can be used to withhold datagrams that arrive out of order. This feature is especially useful in combination with the delivery layer, which does not preserve ordering within the retransmitted messages.

The ordering application preserves ordering by storing packets with a wrong sequence number until all previous messages have arrived. Since messages can be lost, two further functions have been added to the implementation, which prevents starving the following layers if no packets arrive for an extended time. First, a maximum for the number of withheld packets has been added, which allows to deliver all waiting packets in case many packets are arriving before the missing one could be received. When this maximum has been reached, all messages are delivered, disregarding the correct sequence numbers.

The second function is a timer-based delivery. Each time a packet is received, a timer is set, which triggers the delivery of all pending messages on timeout. All pending messages are delivered, in case the previous messages are lost and the sender has stopped sending messages. Listing 6.13 shows the implementation of the `receive_datagram` function.

```
ordering_header header;
binary_deserializer source(writer.system(), received);
if (auto err = source(header))
  // error
if (header.sequence == seq_read_) {
  ++seq_read_;
  cancel_timeout(writer, seq_read_);
  if (auto err = application_.handle_data(
        writer, make_span(received.data() + ordering_header_size,
                          received.size() - ordering_header_size)))
    // error
  return deliver_pending(writer);
} else if (is_greater(header.sequence, seq_read_)) {
  return add_pending(writer, received, header.sequence);
}
```

Listing 6.13: Implementation of the `receive_datagram` function of the ordering application

Sending datagrams through the ordering application is much simpler. The only task the application has to accomplish is the addition of a sequence number. Thus, a header is added to the datagram, which is then passed to the underlying layer. This routine is shown in Listing 6.14.

```
auto hdr = parent.next_header_buffer();
binary_serializer sink(parent.system(), hdr);
if (auto err = sink(ordering_header{seq_write_++}))
  // error
```

```
parent.write_packet(hdr, buffers...);
```

Listing 6.14: Implementation of the `write_message` function of the ordering application

Slicing

A slicing layer has been included in the implementation, for the use with datagram oriented transport protocols. This layer has the task of slicing large datagrams into feasibly large chunks that can be transmitted over networks with a small Maximum Transmission Unit (MTU). Furthermore, UDP, for example, has a fixed maximum datagram size of 65.535 bytes, which will drop data when the amount of data exceeds this boundary.

The implementation of the slicing layer focussed on two things: Slicing outgoing datagrams when necessary and reassembling received datagrams. For this, a slicing header was introduced that specifies the total number of slices and the individual number of each slice that arrived. Outgoing messages are represented in the form of multiple buffers, which requires concatenating them before further processing. The concatenated packet is then sliced into fragments of a configurable `max_fragment_size` size and preceeded with a header.

Reassembling such fragmented datagrams requires storing the data in a queue that holds each fragment until they have all been received. In that case, the fragments are all concatenated in a single buffer and passed on to the next layer.

However, this implementation introduces a very significant runtime-overhead, due to the frequent copying for concatenating buffers. It should be redesigned and implemented without the copying before it can be used for time-sensitive software.

7 Evaluation

In Chapter 5, design goals have been set that will be evaluated in this chapter. Thus, this chapter proposes benchmarks that aim to provide valuable insight to the performance of the new design. However, before showing the benchmark results, efforts to validate the design are presented, which have been implemented along the stack itself.

7.1 Validation

Before the implementation of the new network stack design can be evaluated, it has to be validated. The correct functionality of the implementation is vastly important if performance measurements should be done. Bugs in the implementation can distort the results, which has to be prevented to generate comparable results. Thus, this section shows the steps that have been taken to ensure that the implementation is correct and behaves as intended.

CAF bundles a test-framework called `libcaf_test`, which can be used to implement tests for it. With this framework, the implementation of the network stack has been thoroughly unit and system-tested. Each component in the stack has a separate unit-test, which ensures that the implementation is correct and valid. Furthermore, for the correctness of the whole system, some test-cases for more complex compositions have been added.

7.1.1 Unit Testing

Unit testing is an approach with which every unit of a larger system is tested by itself to ensure its correct functionality. It enables finding bugs very early in the implementation phase, which makes fixing them easier. In CAF most of the components are coupled to a specific unit-test, that ensures their correct functionality.

Listing 7.1 shows the test-case of the doorman. The doorman has the tasks of accepting incoming TCP connections, creating endpoint managers for them, and adding them to the multiplexer. The test-case checks this behavior by creating an endpoint_manager and adding it to the test-multiplexer. After that, a connection to the port of the endpoint manager is established, which should be accepted. Since the doorman will add the newly created endpoint_manager to the multiplexer, the result can be checked by verifying the number of socket-managers in the multiplexer.

```
CAF_TEST(doorman accept) {
  auto acc = unbox(make_tcp_accept_socket(auth, false));
  auto port = unbox(local_port(socket_cast<network_socket>(acc)));
  auto acceptor_guard = make_socket_guard(acc);
  CAF_MESSAGE("opened acceptor on port " << port);
  auto mgr = make_endpoint_manager(mpx, sys,
    doorman<application_factory>{acceptor_guard.release(),
                                 application_factory{}});
  CAF_CHECK_EQUAL(mgr->init(), none);
  auto before = mpx->num_socket_managers();
  CAF_CHECK_EQUAL(before, 2u);
  uri::authority_type dst;
  dst.port = port;
  dst.host = "localhost"s;
  CAF_MESSAGE("connecting to doorman on: " << dst);
  auto conn = make_socket_guard(
    unbox(make_connected_tcp_stream_socket(dst)));
  CAF_MESSAGE("waiting for connection");
  while (mpx->num_socket_managers() != before + 1)
    run();
  CAF_MESSAGE("connected");
}
```

Listing 7.1: Test-case for the correct functionality of the doorman

Since this test relies on the use of networking facilities, this test-case would not be deterministic. Hence, a test-multiplexer is used, which does not run in a dedicated thread. This allows to control the triggering of events manually, which allows for deterministic test-cases.

7.1.2 System Testing

Unit-tests are a valuable approach to testing components individually, but testing compositions of them is equally important. In the resulting software system, the components are used in a composed way, which cannot be verified individually. Thus, system-tests are added as well, to verify the correct implementation of the composed system.

These tests aim to check actual actor communication using the network backend. For example, a ping pong test-case is included, which tests the resolving process, sending, and receiving actor messages using the TCP backend. With such test-cases the interoperability of the components is checked and verified for the future altering of components.

7.2 Benchmarks

The evaluation of the new network stack in CAF considers two performance metrics: The throughput and the latency. For this, two benchmarks are proposed, which simulate different types of work-loads for the stacks so that the performance can be measured. For the throughput measurements, a streaming-benchmark is proposed, which transmits large amounts of data between many nodes. For the latency measurements, a ping pong benchmark is proposed, which exchanges a number of messages and waits for the answer of the remote node.

All benchmarks in this section have been conducted on a machine with 128 cores and 512GB RAM running Ubuntu 18.04. The configuration of the actor systems has been tuned, so that exactly a single thread for the scheduling of actors and another one for the I/O backend are spawned. Thus, the benchmarks have been executed with configurations up to 64 remote nodes that participate in the communication. Furthermore, for the communication real TCP sockets have been used, which communicate over the localhost interface.

7.2.1 Throughput of the Network Stack

Throughput was one of the performance goals, that have been set in Chapter 5. To measure the throughput, a streaming-benchmark is proposed that uses the network layer to stream data concurrently to a number of remote nodes. However, measuring

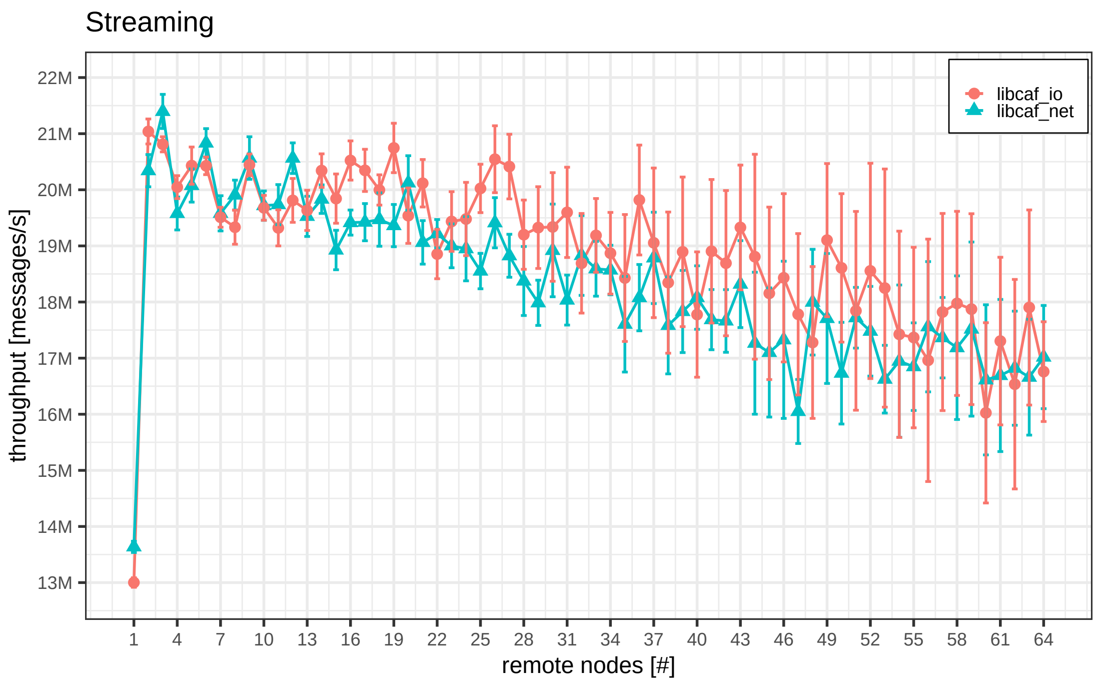


Figure 7.1: Streaming benchmark results for 1 to 64 concurrently served remote nodes

the throughput by itself is not sufficient, since it does not take the scaling capabilities of the stack into consideration. Thus, the benchmark gradually increases the number of remote nodes, which the data is sent to.

The data that is sent consists of a sequence of `uint64_t` (8 byte) integer values. These values are counted on the remote node and the total number is saved every second, which is possible because the values are passed to the remote actor individually. On each node, one stream source was spawned which communicates with a stream sink on the central node via a TCP socket.

In Figure 7.1, the results for the streaming benchmark are shown for `libcaf_io` and `libcaf_net`. The x-axis maps the number of remote nodes and the y-axis shows the accumulated number of values that have been received every second. The graph shows that `libcaf_net` has similar capabilities in terms of throughput compared to `libcaf_io` . Both designs have a maximum throughput of about 21 million values, which is what was aimed for in Chapter 5.

Noteworthy about the result is the fluctuation of the throughput of `libcaf_io`. In comparison, `libcaf_net` shows a much more stable behavior, especially with more than 30 remote nodes.

7.2.2 Latency of the Network Stack

The latency of the new design has been evaluated with a ping pong benchmark that exchanges messages between a local and a remote node. Since actors can only react upon receiving a message, this type of benchmark is well-suited for the evaluation of the latency. The duration of the processing on both nodes directly affects the number of round-trips that can be fulfilled within a certain time. Thus, the benchmark is capable of capturing the latency of the stack.

The proposed ping pong benchmark consists of two types of actors: ping and pong actors. Ping actors send ping-messages to a remote node, which are answered by pong-actors with a pong-message. Upon receiving pong-messages, the ping-actor increments a counter, which is saved and reset every second.

To be able to evaluate the scaling-capabilities of both stacks in such a setup, the number of participating pong-actors is gradually increased. This increases the number of sockets that have to be managed by the multiplexer and simulates a more realistic workload for it.

Figure 7.2 shows the result for a simple pingpong configuration, scaled across 1-64 nodes. In this setup, the ping-actor sends a single ping-message to each remote node. The x-axis shows the number of remote nodes that participate in the communication, while the y-axis shows the accumulated number of received pong-messages per second. Each pong is essentially a representation for a round-trip between the ping and a pong actor.

The results show that `libcaf_net` poses problematic behavior for both UDP and TCP. The total number of pongs starts to converge between 40-50 thousand round-trips per second, which is about half of the amount `libcaf_io` is capable of. This result shows that the latency of the implementation in `libcaf_net` is much higher than the latency in `libcaf_io`. The reason for this behavior however, cannot be analyzed with a pingpong benchmark, hence an in-depth analysis is required to find the reason behind the issue with the latency.

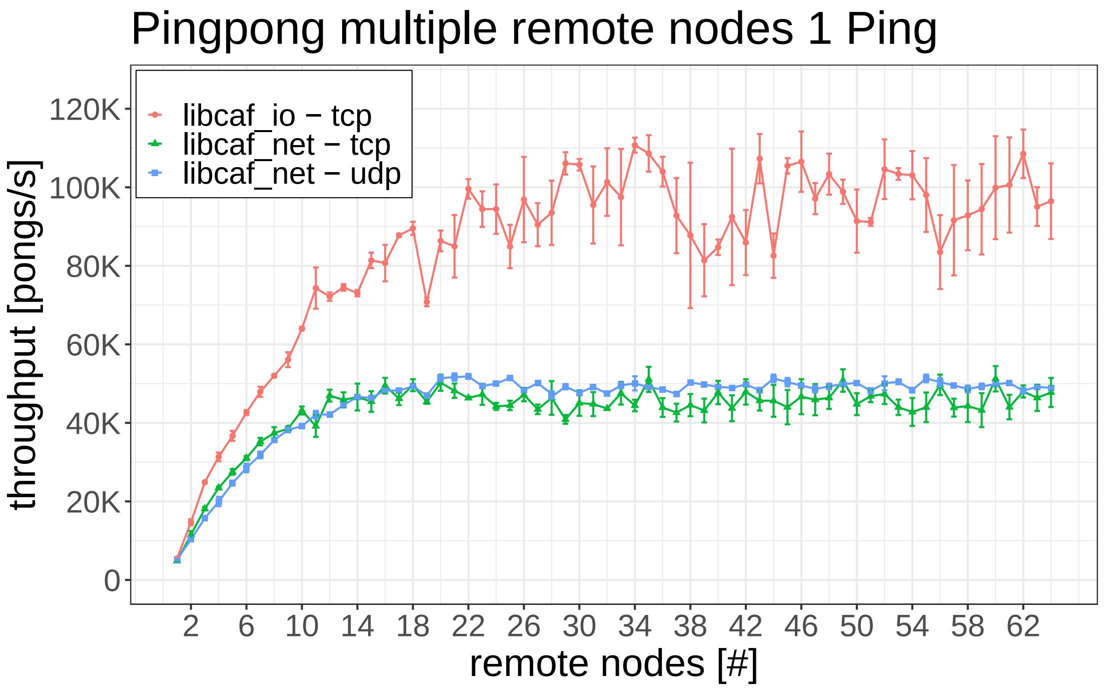


Figure 7.2: Ping-pong benchmark with a single ping message per endpoint

What can be observed however, is that the reason most-likely has nothing to do with the protocol in use. Both UDP and TCP behave very similarly and converge at around the same number of roundtrips.

Comparing Processing-Overheads of `libcaf_io` and `libcaf_net`

To achieve a better understanding of what impairs the latency of the new design, a more precise analysis is necessary. The ping pong benchmark offers a good first impression but hides a lot of valuable information. Thus, the next step is to identify the task that requires most of the time, to be able to identify the issue.

This has been done by taking timestamps throughout the stack for sending and receiving messages to and from a remote node using TCP. Both `libcaf_io` and `libcaf_net` have been taken into consideration, so that a comparison of both results could be done. Figure 7.3 plots the results of the measurements.

The plot shows that the processing of incoming and outgoing messages takes significantly longer in `libcaf_net`. Processing incoming messages requires about 30μs longer, while

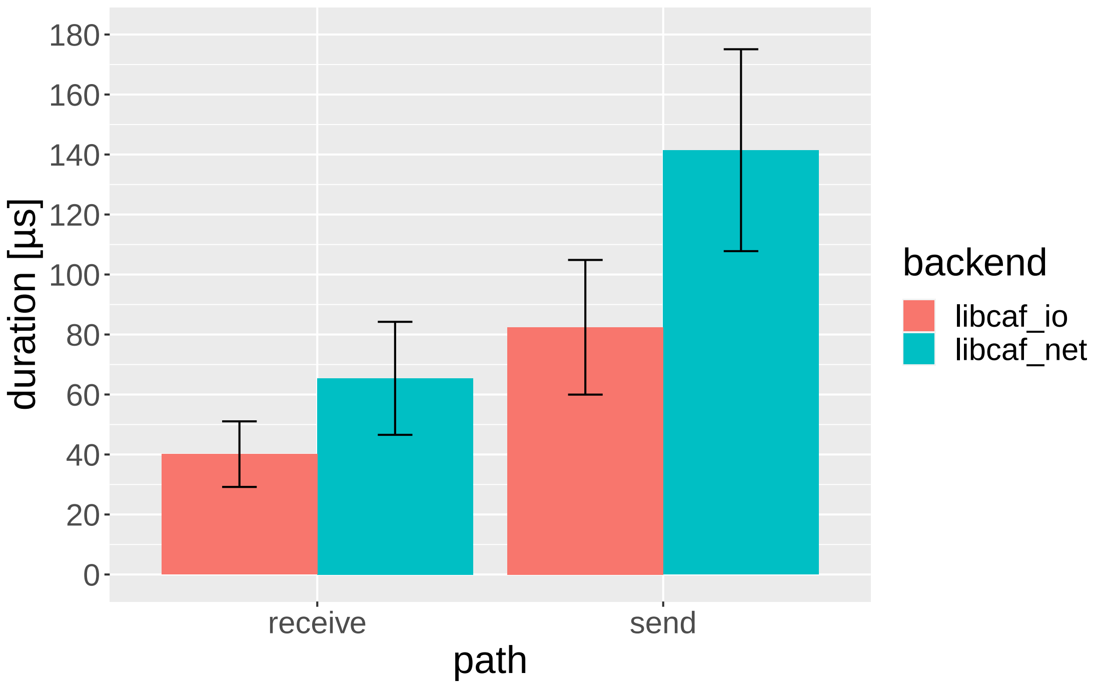


Figure 7.3: Durations of processing messages in both the old design (`libcaf_io`) and the proposed design (`libcaf_net`)

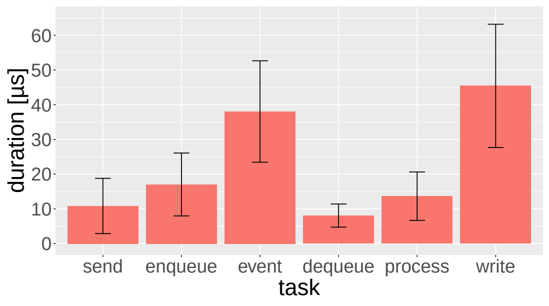
processing outgoing messages requires even more time with over 60μs more. This shows that there must be a bug in the implementation of `libcaf_net`, which has to be investigated further.

Detailed Analysis of the Processing-Overhead in `libcaf_net`

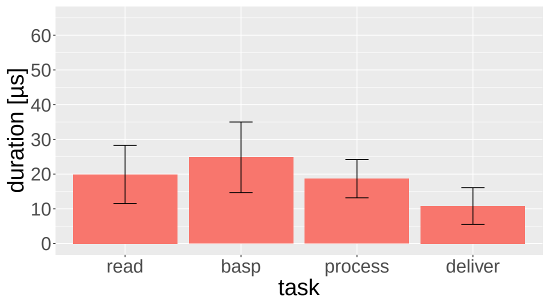
The approach of taking timestamps has been repeated in more detail, which leaves the results incomparable to `libcaf_io`. Since the implementations differ greatly, the following results only show `libcaf_net`. For this analysis, one message per second was sent to a remote node, which ensures that the stack is in an idle state at the time of taking timestamps. Thus, allowing to capture the worst-case behavior of the stack.

Figure 7.4a shows the detailed results for processing outgoing messages. For this, the stack was divided into six individual section, which are divided as follows:

send This section captures the `send` call of the actor, which creates a message, and passes it through the actor proxy to the endpoint manager.



(a) Outgoing messages



(b) Incoming messages

Figure 7.4: Detailed analysis of durations in the new network stack design

enqueue This section captures the enqueuing of the message to the endpoint manager.

event This section shows the duration that the multiplexer requires until a socket-event is triggered. The overhead of this section can not be avoided as it is imposed by the `poll` implementation in the OS.

dequeue This section shows the dequeuing of a message from the endpoint manager.

application This section includes threefold: Traversing the application stack, processing the message and enqueuing it to the transport abstraction.

write This section includes the process of writing a single message to the socket.

In the analysis, a positive result can be observed: The application stack abstraction does not introduce noticeable overhead. Traversing the stack and processing outgoing messages combined requires an expected but in no way excessive amount of time.

However, three sections in the stack introduce an unexpectedly large overhead, which has to be analyzed further. First, the process of enqueuing and dequeuing messages to and from the endpoint manager both require significant durations. Multiple actors can concurrently enqueue messages to the queue, which can lead to race-conditions. Thus, the access has to be synchronized which adds overhead to the actual process of enqueuing messages. Furthermore, the whole queue is built using nodes that have to be heap-allocated. This approach introduces a second performance overhead, which is actually not necessary.

Initially, the message-queue was introduced for storing the message and its serialized counterpart, since the messages have been serialized in the actor proxy. Due to a design change, the serialization was moved to the application stack, without altering the queue.

However, actor messages themselves can be linked together to form such a linked-list, which leads to the queue unnecessarily allocating memory that is not actually needed.

The second critical section is the writing of data to sockets. It requires a very significant amount of time, which is introduced by the multiple system-calls required for writing multiple buffers to the socket. The new design proposed a scatter-gather I/O API, which allows scattering messages across multiple buffers to avoid copying data. However, the implementation of the transport abstraction writes each buffer sequentially, which takes at least two `write`-calls per message.

Figure 7.4b shows the results of the timing-analysis for receiving messages. The whole process of processing incoming messages is much simpler than sending them. However, the path of a single message could still be separated into several sections:

read This section includes the read call for the header on the socket.

basp This section includes passing the received message to BASP and starting a worker for processing it.

process This section includes the processing of the received message.

deliver This section includes delivering the message to the recipient.

The figure shows that the way through the BASP application and the processing both take a significant amount of time. This is due to the approach of using deserializing workers, which concurrently deserialize messages before delivering them. Implementing this process concurrently introduces a performance overhead because both acquiring a worker instance and starting it are expensive tasks. Synchronization and copy overheads add up and form a significant overhead. However, this approach is also used in `libcaf_io`, which shows that this approach should be much more performant. Most likely, the implementation in `libcaf_net` has a bug that impairs the performance in this way.

7.2.3 Discussion

The evaluation of the new design has shown that the goals of realizing a stackable application abstraction layer could be met. Application protocols can be included and composed in any way, which allows building complex network stacks. Especially the measurements with UDP have shown that the composable design works well. The precise measurements of the stack in Figure 7.4 have shown that the introduced performance overhead by the layering approach is very small.

Furthermore, the throughput goal for the design could be met, which was shown in the streaming-benchmark. The results in Figure 7.1 show that the implementation of the new design is as performant as the current implementation in `libcaf_io`.

However, the implementation of the design still has issues that have to be fixed before the stack can be used as the default network backend for CAF. These issues manifest themselves in the form of very high latency which can be explained with three reasons:

1. The message queue introduces significant runtime-overhead when enqueuing and dequeuing messages.
2. The process of sending messages via TCP is implemented imperformantly.
3. Processing and delivering incoming messages using concurrent workers is imperformant for small loads.

The message queue that is used by the endpoint manager is a key feature of the design that decouples actors from the network layer. Outgoing messages are stored in the queue in a singly-linked list, which uses heap-allocated buckets to store the messages. This heap-allocation introduces an expensive system-call for every message that should be stored, which is very imperformant.

Furthermore, the approach is not required because the actor-messages are of the same type as the buckets, which renders the allocation unnecessary. The design initially proposed to serialize messages in the actor-proxy before enqueuing them, which was remodeled in the implementation phase. However, the message-queue was not refactored accordingly, which is a serious flaw in the implementation that has to be reconsidered.

The second issue that has been uncovered, was the implementation of writing data to sockets. With the proposal of scatter-gather I/O, each message is passed to the transport layer in multiple buffers, which are stored individually to mitigate the copy overhead for concatenating them. However, because of this, multiple write-calls for each message are required until all buffers are written. This adds a significant overhead to the transmission process, which could otherwise be avoided.

Two solutions are proposed for future approaches: Using `writev` instead of `write` or giving up the scatter-gather approach in-favor of the single buffer approach `libcaf_io` is based on. Giving up the scatter-gather approach has the disadvantage of complicating the processing of messages on the application layer. Writing headers before packets will introduce a copy overhead to the message-processing, which is undesirable and thus not feasible.

Using `writev` for writing data to the socket is much better suited. It is an OS feature that allows writing multiple buffers in a single call, which avoids multiple system-calls. With this approach, the added overhead could potentially be avoided.

The third issue is the latency between the processing of incoming messages. The concurrent deserializing of the messages requires entering multiple synchronized critical sections, which adds a runtime-overhead. However, since this approach has also been used in the implementation of `libcaf_io`, the most feasible reason for the overhead is a bug in the implementation. The implementation of the process should thus be revalidated and reevaluated before using the stack in CAF.

8 Conclusion and Future Work

The current design of the network layer in CAF largely grew with the requirements for it. In it, the transport protocol abstraction has always been tightly coupled to the stream-oriented model of TCP. Transmission guarantees as well as the detection of the liveliness of remote nodes were simply inherited from the protocol and hence, never implemented. Furthermore, the inclusion of application protocols is a requirement that has emerged but was never implemented in a way that could be used in CAF. This shows that the current network layer abstraction is very limited in both reliability and extensibility.

This thesis evaluated the problems with the current state and formulated new requirements for a reliable network layer abstraction. A new design has been proposed that aimed to solve the problems with the current design, while also including the new requirements. With the redesign, the strong coupling to TCP was removed, which allows exchanging it for any other protocol. Guarantees that have previously been inherited by the transport protocol can now be added as transport extensions. This allows extending the guarantees and functions of transport protocols and adds versatility to the overall design.

The application layer was redesigned with the goal of introducing composability to it. The proposed solution in this thesis relies on application-protocol instances that can be layered and thus composed in any way. This addition allows building complex compositions of network stacks fast and simple, with reusable components.

A number of application protocols has been included in the design, which are designated to adding reliability options to simpler transport protocols such as UDP. These protocols include ordering and delivery options that allow adding reliability features when necessary. Furthermore, a slicing layer has been proposed that handles slicing large messages into smaller chunks for the transmission over UDP

The process of resolving actors and connecting to remote nodes could simplified significantly by introducing a URI-based resolving process. With this addition, publishing actors can now be accomplished by name rather than binding them to ports of the machine. It furthermore allowed including more expressive error-categories, which ease the search for bugs during this process. Thus, making distributing software based on CAF more robust and simpler to use.

This thesis extensively discussed issues with the current design and proposed solutions for most of them. Furthermore, a new network layer design for CAF was proposed and implemented. However, there are still topics that could not be covered. The following list shows the topics that will be left for future considerations:

UDP The implementation of the `datagram_transport` is in working condition and can be used to communicate with remote nodes. Multiplexing the single socket for communicating with multiple remote endpoints however, still poses problems. Furthermore, closing the communication and purging the local connection-state has not been implemented.

Transport Protocols Currently the design only provides transport abstractions for UDP and TCP. Designs for other transport protocols such as SCTP or QUIC have been concluded, but not implemented. Hence, these protocols should be implemented and added to the implementation.

Application Protocols Including application protocols is now easily possible, but only a limited number of them have been implemented. HTTP, WebRTC, ICE, and more, would be great additions to the new design.

Encryption Encryption is a topic that has been covered in the design but has not been implemented.

Performance The state of the current implementation leaves much room for improvement. Especially the evaluation of the latency showed that some bugs might still be present. Especially, the design of queuing messages and writing data to the sockets has to be reconsidered to solve the issue.

Documentation The public API of the implementation has been thoroughly commented and annotated. However, there is no manual for how to use the new stack. Before the design can be added to CAF, this step should be fulfilled.

Bibliography

[1] D. Charousset, R. Hiesgen, and T. C. Schmidt, “Revisiting Actor Programming in C++,” *Computer Languages, Systems & Structures*, vol. 45, pp. 105–131, April 2016.

[2] J. Armstrong, *Making Reliable Distributed Systems in the Presence of Software Errors.* PhD thesis, Department of Microelectronics and Information Technology, KTH, Sweden, 2003.

[3] R. Hiesgen, D. Charousset, and T. C. Schmidt, “OpenCL Actors—Adding Data Parallelism to Actor-based Programming with CAF,” in *Programming with Actors - State-of-the-Art and Research Perspectives* (A. Ricci and P. Haller, eds.), no. 10789 in Lecture Notes on Computer Sciences (LNCS), pp. 59–93, Berlin, Heidelberg, N.Y.: Springer Verlag, 2018.

[4] J. Postel, “Transmission Control Protocol,” RFC 793, IETF, September 1981.

[5] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” Internet-Draft – work in progress 29, IETF, June 2020.

[6] J. Postel, “User Datagram Protocol,” RFC 768, IETF, August 1980.

[7] C. Hewitt, P. Bishop, and R. Steiger, “A Universal Modular ACTOR Formalism for Artificial Intelligence,” in *Proc. of the 3rd IJCAI*, (San Francisco, CA, USA), pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.

[8] G. Agha, *Actors: A Model of Concurrent Computation In Distributed Systems.* Cambridge, MA, USA: MIT Press, 1986.

[9] J. Armstrong, “A History of Erlang,” in *Proc. of the third ACM SIGPLAN conference on History of programming languages (HOPL III)*, (New York, NY, USA), pp. 6–1–6–26, ACM, 2007.

[10] Lightbend Inc., "Akka Framework." `http://akka.io`, August 2020. Accessed: 2020-05-23.

[11] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: Cloud Computing for Everyone," in *Proc. of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, (New York, NY, USA), pp. 16:1–16:14, ACM, 2011.

[12] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch, "Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments," in *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!*, (New York, NY, USA), pp. 87–96, ACM, Oct. 2013.

[13] J. Torrellas, H. S. Lam, and J. L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches," *IEEE Trans. Comput.*, vol. 43, pp. 651–663, June 1994.

[14] The Open MPI Project, "Open MPI: Open Source High Performance Computing." `https://www.open-mpi.org/`, January 2020.

[15] M. Aron and P. Druschel, "TCP Implementation Enhancements for Improving Webserver Performance," in *INFOCOM 1999*, 1999.

[16] M. Karol, M. Hluchyj, and S. Morgan, "Input versus output queueing on a space-division packet switch," *IEEE Transactions on communications*, vol. 35, no. 12, pp. 1347–1356, 1987.

[17] J. Postel and J. Reynolds, "Telnet Protocol Specification," RFC 854, IETF, May 1983.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, IETF, June 1999.

[19] T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Transport Layer Protocol," RFC 4253, IETF, January 2006.

[20] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The ϕ accrual failure detector," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pp. 66–78, IEEE, 2004.

[21] M. Cox, R. Engelschall, S. Henson, B. Laurie, and t. al. t. al., "OpenSSL." http://www.openssl.org, 2008.

[22] C. Amsuess, “OSCORE Implementation Guidance,” Internet-Draft – work in progress 00, IETF, April 2020.

[23] D. Peleg and E. Upfal, “The generalized packet routing problem,” *Theoretical Computer Science*, vol. 53, no. 2-3, pp. 281–293, 1987.

[24] J. Postel, “Internet Protocol,” RFC 791, IETF, September 1981.

[25] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” *Computational Science and Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[26] R. Hiesgen, D. Charousset, and T. C. Schmidt, “A Configurable Transport Layer for CAF,” in *Proc. of the 9th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH ’18), Workshop AGERE!*, (New York, NY, USA), pp. 1–12, ACM, Nov. 2018.

[27] Hiesgen, Raphael, “Redesigning the Network Layer for Distributed Actors in CAF,” Master’s thesis, Hochschule für Angewandte Wissenschaften Hamburg, November 2017. `https://inet.haw-hamburg.de/thesis/completed/raphael-hiesgen/@@download/file/MA_raphael_hiesgen.pdf`.

[28] J. Rosenberg, “Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols,” RFC 5245, IETF, April 2010.

[29] D. C. Schmidt, “The ADAPTIVE Communication Environment: An object-oriented network programming toolkit for developing communication software,” in *11th and 12th Sun Users Group Conference*, 1993.

[30] B. Trammell, M. Welzl, T. Enghardt, G. Fairhurst, M. Kuehlewind, C. Perkins, P. Tiesel, C. Wood, and T. Pauly, “An Abstract Application Layer Interface to Transport Services,” Internet-Draft – work in progress 07, IETF, July 2020.

[31] T. Pauly, B. Trammell, A. Brunstrom, G. Fairhurst, C. Perkins, P. Tiesel, and C. Wood, “An Architecture for Transport Services,” Internet-Draft – work in progress 08, IETF, July 2020.

[32] A. Brunstrom, T. Pauly, T. Enghardt, K.-J. Grinnemo, T. Jones, P. Tiesel, C. Perkins, and M. Welzl, “Implementing Interfaces to Transport Services,” Internet-Draft – work in progress 07, IETF, July 2020.

[33] T. Berners-Lee, R. Fielding, and L. Masinter, “Uniform Resource Identifier (URI): Generic Syntax,” RFC 3986, IETF, January 2005.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „- bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] - ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: §16 Abs. 5 APSO-TI-BM bzw. §15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: ________________________________

Vorname: ________________________________

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Redesigning and Evaluating the Network Stack in the C++ Actor Framework

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original