Master Thesis

Raphael Hiesgen

Redesigning the Network Layer for Distributed Actors in CAF

Raphael Hiesgen

Redesigning the Network Layer for Distributed Actors in CAF

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master 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. Michael Köhler-Bußmeier

Eingereicht am: November 13, 2017

Raphael Hiesgen

Thema der Arbeit

Redesigning the Network Layer for Distributed Actors in CAF

Stichworte

Verteilte Systeme, Aktormodell, Nebenläufigkeit

Kurzzusammenfassung

Das Aktormodell baut auf transparenter Nachrichtenübermittlung und einem starken Fehlermodell auf. Es verspricht Skalierbarkeit über nebenläufige Kerne und verteilte Knoten. Hochverteilte Systeme, die skalierbar auf Anwender oder Verarbeitungsanforderungen abgestimmt sind, können von dem hohen Abstraktionsniveau von Aktoren profitieren. Das C++ Actor Framework (CAF) ist eine native Implementierung des Aktormodell, die einen effizienten Nachrichtenaustausch beinhaltet. In dieser Arbeit analysieren wir die Verteilungsschicht in CAF mit dem Ergebnis, dass unterschiedliche Leistungsgarantien in lokalen und verteilten Umgebungen die geforderte Verteilungstransparenz beeinträchtigen. Wir diskutieren die Aspekte Zuverlässigkeit, Erreichbarkeit und Bindung an ein Transportprotokoll im Kontext der Aktorkommunikation, bevor Garantien für die Kommunikation im CAF abgeleitet werden. Auf dieser Grundlage schlagen wir eine Neugestaltung der CAF-Netzwerkschicht vor. Diese erlaubt den Austausch von Transportprotokollen, definiert eine konsistente Auswahl von Garantien für Nachrichtenübertragung zwischen Aktoren und entfernt Routing-Fähigkeiten aus dem CAF-Overlay. Die Erweiterung der CAF-Netzwerkschicht um UDP bietet eine erste Bewertung der Funktionalität und ebnet den Weg für eine vollständige Implementierung des vorgeschlagenen Designs.

Raphael Hiesgen

Title of the paper
Redesigning the Network Layer for Distributed Actors in CAF

Keywords
Distributed systems, actor model, concurrency

Abstract
The actor model offers network transparent message passing and a strong failure model. It promises scalability over concurrent cores and distributed nodes. Highly distributed systems designed to scale with users or processing demands can benefit from its high level of abstraction. The C++ Actor Framework (CAF) is a native implementation of the actor model with an efficient message passing layer. In this work, we examine the actor communication in CAF and find distribution transparency to be impacted by varying guarantees for concurrency and distribution. The aspects of reliability, reachability and transport binding are reconsidered in the context of actor communication before deriving guarantees for the communication in CAF. To address these differences, we propose a redesign of the CAF network layer. This introduces exchangeable transport protocols, defines a consistent set of guarantees for message passing between actors and removes routing capabilities from the CAF overlay. The augmentation of the CAF network layer with UDP provides a first assessment of the functionality and paves the way for a complete implementation of the proposed design.

Contents

List of Tables

List of Figures

Listings

1 Introduction

The actor model and its implementations have recently received much attention. Actor systems promise transparent concurrency and distribution by combining message passing with a strong failure model. The demand for scalable systems rises across the industry.

Data centers process large amounts of data to extract information in near realtime while backend servers for massive multiplayer online games scale with the number of active users to provide fluent interaction between players across the world. Scaling down to low-end devices, the Internet of Things (IoT) is steadily growing and collecting data in cloud backends. The IoT spans a distributed system that consist of low-powered embedded hardware produced at minimal cost. Network and synchronization primitives are often hand-crafted, leaving room for errors that are hard to correct after deployment. A common characteristic among these applications is a highly distributed environment that necessitates scalability with the number of participants. The actor model addresses this need and offers a high level of abstraction to design and develop scalable applications.

Developers for all environments can benefit from well-tested and robust frameworks that enable application development at a high abstraction level and allow them to focus on the application logic instead of re-inventing low-level communication and synchronization primitives. Viewing actors as a scalable solution to design and develop distributed systems, frameworks have to address the various deployment environments and their respective network characteristics.

The C++ Actor Framework (CAF) [1] is a lightweight implementation of the actor model in modern C++. It combines a high level API with an efficient message passing layer. Much work has been put into the implementation of a scalable scheduler and robust message passing interfaces that allow type checking at compile time. The network stack of the framework and the application layer protocol used to manage distributed CAF nodes mostly evolved with the demands. The requirements for communication in distributed CAF-based systems have not yet been defined. Thus, there is no set of guarantees to rely on for actor-to-actor communication. As an essential component for distribution, the design and capabilities of the network stack affect general behavior of applications and influence scalability.

The current network stack is tightly coupled to TCP and relies on the transport protocol for its messaging guarantees. Like in most actor systems, the behavior of CAF under distribution differs from local concurrency. For example, message passing to remote actors is unreliable while messages are reliably delivered to local actors. In general, reliability in distributed systems is a major source of complexity and much harder to achieve than in local regimes. There is a need to address these discrepancies and define the communication requirements in distributed actor systems. Thereafter, a set of guarantees for actor communication in CAF can be derived and documented clearly. These considerations provide the opportunity to examine not only reliability but include general messaging aspects such as security, reachability and transport bindings.

Transport bindings especially, should fit their deployment scenario. While TCP is the dominant protocol throughout the Internet, HTTP tunnels and WebRTC can enable communication between nodes hidden behind firewalls and NATs. Scaling to high performance environments, technologies such as InfiniBand enable high throughput in closely coupled clusters. On the other end of the scale, constrained environments depend on specialized standards such as 6LoWPAN or CoAP over UDP to address lossy networks. Choosing a transport protocol is a trade off between the scope of services a protocol offers in contrast to lightweight and efficient data exchange. Simply deploying the protocol that offers the most guarantees is not a viable solution. While constrained environments might not be able to handle the messages sizes or network load, other applications might require low-latency and would rather loose messages than wait for retransmits. This tradeoff further motivates the need for an exchangeable transport layer to address scalability as well as dynamic deployment.

In this work, we reconsider communication aspects of distributed actor systems using the C++ Actor Framework as an application domain. In search for enhanced transparency, we question whether a lightweight system can provide the same behavior for distributed as for concurrent scenarios. There is much to gain from such a system. Developing, testing and arguing about the behavior of the system becomes much easier if its behavior does not depend on the deployment. Many guarantees are cheap to establish locally, but vastly increase in cost when network communication is introduced. A weaker requirement would be the detection of deviating behavior by the runtime system. While not as strong a guarantee, developers would be made aware of deviating behavior.

This thesis is organized as follows. Chapter 2 introduces the actor model and presents the C++ Actor Framework, a native implementation with a focus on scalability and efficiency. Subsequently, Chapter 3 discusses the challenges that arise from designing and implementing a distributed systems that scales at large. The aspects reliability, reachability, scalability and security are discussed in detail in Chapter 4, including related work such as standard solutions and contributions from other actor systems. An extended design of the CAF network stack is proposed in Chapter 5 before discussing an implementation in Chapter 6. Chapter 7 presents tests of the newly integrated network components. Finally, a conclusion is drawn in Chapter 8.

2 The Actor Model of Computation

With the advent of multicore machines and cloud computing, the actor approach has gained momentum over the last decade. In this environment, scalability and fault-toleranency are important traits. This section presents the actor model in general before it discusses a specific implementation of the actor model, the C++ Actor Framework [1] which is the foundation of this thesis.

2.1 Actors for Concurrency and Distribution

The actor model defines entities called actors. Actors are concurrent, isolated entities that interact via message passing. They use unique identifiers to address each other transparently in a distributed system. In reaction to a received message, an actor can, (1) send messages to other actors, (2) spawn new actors and (3) change its own behavior to process future messages differently. These characteristics lead to several advantages. Since actors can only interact via message passing, they never corrupt each others state and avoid race conditions by design.

The operation to create new actors is called `spawn`. It is often used to distribute workload, e.g., in a divide and conquer approach. An actor divides a problem and spawns new actors to handle the subproblems concurrently. Thereby, the created actors can divide and distribute their problems further.

To detect and propagate errors in local as well as distributed systems, actors can monitor each other. When a monitored actor terminates, the runtime environment sends a message containing the exit reason to all monitoring actors. A stronger coupling of lifetime relations can be expressed using bidirectional links. In a linked set of actors, each actor will terminate with the same error code as its links. As a consequence, links form a (sub-) system in which either all actors are alive or have failed collectively. This allows developers to re-deploy failed parts of the system at runtime and prohibits invalid or intermediate states.

Hewitt et al. [2] proposed the actor model in 1973 as part of their work on artificial intelligence. Later, Agha formalized the model in his dissertation [3] and introduced mailboxing for processing actor messages. He created the foundation of an open, external communication [4]. At the same time, Armstrong took a more practical approach by developing Erlang [5].

Multiple actor-based languages have been developed in the last decades. A typical example is the Erlang programming language introduced by Armstrong [5]. It was designed to build distributed system that run without downtime and originally targeted telephony applications. Actors are included in Erlang in the form of processes of the same characteristics as actors. Erlang provided the first de-facto implementation of the actor model, despite using a different vocabulary. Scala is a programming language that includes actors as part of its standard distribution through the Akka framework [6]. It offers object oriented as well as functional programming and runs in Java virtual machines (JVM).

2.2 Native Actors in C++

The C++ Actor framework (CAF) [7] combines the benefits of native program execution with a high level of abstraction. The best known implementations of the actor model, Erlang and Akka, are both implemented in languages that rely on virtual machines. In contrast, CAF is implemented in C++ and thus compiles to in native code. C++ is used across the industry from high performance computing installations running on thousands of computing nodes all the way down to systems on a chip. With CAF, we propose a C++ framework to fill the gap between the high level of abstraction offered by the actor model and an efficient, native runtime environment.

Following the tradition of the actor model, actors are created using `spawn`. The function takes a C++ functor or class and returns a handle to the created actor. Hence, functions are first-class citizens and developers can choose whether they prefer an object-oriented or a functional software design. Per default, actors are sub-thread entities scheduled cooperatively using a work-stealing algorithm [8]. This results in a lightweight and scalable actor implementation that does not rely on system-level calls, e.g., required when mapping actors to threads. Uncooperative actors that require access to blocking function calls can still be bound to separate threads by the programmer to avoid starvation.

Unlike other implementations of the actor model, CAF differentiates between *addresses* and *handles*. The former is used for operations supported by all actors such as monitoring. The latter is used for message passing and restricts messages to the specified messaging interface in case of strongly typed handles, or allows any kind of message in case of untyped handles. Access to local and remote actors is transparent. There is no differentiation between them on the API level, thus hiding the physical deployment at runtime.

Actors can communicate asynchronously by using `send` or synchronously by using `request` . While the runtime does not block in both cases, `request` lets the user synchronize two actors. When sending such a message, a message handler can be set to handle the response. Then, the sending actor waits until a response is received before processing other incoming messages. Alternatively, a timeout can be specified to return to the previous behavior if no response is received. In case of errors, e.g. when the receiver is no longer available or does not respond, an error message will be sent to the sender and cause it to exit unless a custom error handler is defined.

Messages are buffered in the mailbox of the receiver in order of arrival before they are processed using the designated behavior. A behavior in our system consists of a set of message handlers. These may include a message handler that is triggered if no other message arrives within a declared time frame. Actors are allowed to dynamically change their behavior at runtime using `become` to change to a new behavior or `unbecome` to return to the previous one.

C++ is a strongly typed language that performs static type checking. Building upon this, it is only natural to provide similar characteristics for actors. With typed actors, CAF provides a convenient way to specify the messaging interface in the type system itself. This enables the compiler to detect type violations at compile time and to reject invalid programs. In contrast, untyped actors allow for a rapid prototyping and extended flexibility. Since CAF supports both kinds of actors, developers can choose which to use for which occasion.

CAF has been compared against other actor implementations [9], namely Erlang, the Java frameworks SALSA Lite [10] and ActorFoundry (based on Kilim [11]), the Scala toolkit and runtime Akka [6] and Charm++ [12]. Metrics that were considered are (1) actor creation overhead, (2) sending and processing time of message passing implementations, (3) memory consumption for several use cases. Furthermore, a (4) a benchmark from the Computer Language Benchmarks Game was picked up. The results showed that CAF displays consistent scaling behavior, minimal memory overhead and very high performance.

2.3 Application Domains

Any application meant to scale with the available resources can benefit from implementation in the actor model. A prerequisite is a problem that can be divided into enough independent tasks to benefit from a high amount of parallelization. Prominent examples are applications that scale with user demand, e.g., the backend servers of massive multiplayer online games (MMOs)—as seen in first commercial implementations—as well as applications that benefits from scalability over many cores and nodes to optimize throughput such as data processing applications. VAST (Visibility Across Space and Time) [13, 14] is a network forensic tool build with CAF to processes large amounts of network events and allow interactive queries from users.

The Internet of Things (IoT) consists of very different hardware, but faces similar changes in synchronization and distribution. Instead of capable nodes with many cores, it deploys large distributed systems. Emerging Internet standards enable this huge number of embedded machines to interconnect and cooperate—and raise the challenge of building suitable software environments that provide scalability, reliability, and security at an efficient level. Since traditional programming and runtime environments are too heavy-weight for this constrained environment, many developers fall back to low-level, hardware-specific programming. As a result, code is barely portable, because network communication and synchronization are hand-crafted. This approach inherently has a high complexity and many sources of errors. The actor model offers a suitable approach to developing software for this environment. It provides a high abstraction level to program concurrent systems and includes network transparent message passing, as well as an error model designed for distributed systems [15, 16]. Ongoing efforts to port CAF to the friendly embedded OS RIOT [17] will ease its deployment in IoT environments.

The processing power of modern many core hardware such as graphics processing units (GPUs) or coprocessors is increasingly available for general-purpose computation. The seamless way of actor systems to addresses concurrent and distributed programming makes it an attractive approach to integrate these architectures. CAF offers access to these specialized devices in form of OpenCL-enabled actors. This type of actors offers a high level interface for accessing any OpenCL device without leaving the actor paradigm. Integrated into the runtime environment, they extend transparent message passing in distributed systems to heterogeneous hardware [18].

3 Problem Space

The network stack of CAF is a central part of the framework and naturally affects the systems behavior in a distributed environment. Built from many individual components, the stack has a large design space, ranging from identifiers for individual actors over the protocol used for the communication to security considerations. This chapter discusses a few challenges to consider throughout this work.

3.1 Access Control

Deploying an actor system with a public interface allows messaging from arbitrary endpoints. Using the API function `remote_actor`, a remote runtime can connect to a published actor and start interacting with the system. While typed actor interfaces define messages related to the application logic, management messages are handled by the runtime environment and can be sent independent of the application logic. This way, malevolent systems can spawn new actors to hog resources or simply kill running actors.

Not all environments have a need for access control. There are three different scenarios to consider, differing in the trust relations of the nodes and network. If nodes and network are trusted, access control is not a big concern (precluding compromise of the system). Example scenarios are tightly coupled clusters or servers that interact via a private network interface.

A distributed system of controlled and trusted nodes can rely on an untrusted network for communication. Access control is required to prevent untrusted nodes from joining the distributed system. This can be achieved through a transport protocol that provides authentication such as TLS [19] or DTLS [20]. Instead of integrating such functionality into CAF, implementation should be left to domain specialists. Standard libraries offer a robust and maintained basis for integration into the framework.

Lastly, a system in which neither the network nor its nodes can be trusted requires more than authentication. An example is a public API to interact with a service that is open to third party clients. In addition to authentication, these cases require the runtime environment to discard malicious management messages and thus prevent the creation and termination of local actors. Basically, access to a public service should be restricted to its intended use. CAF allows the creation of actors called brokers that expose a limited API over a different protocol. For example, a broker could offer a REST API based on HTTP and forward valid request as actor messages internally. While this adds implementation and processing overhead, the interface cleanly separates internal from public messaging.

3.2 Efficiency

Efficiency is not a goal addressed by a single component but factors into the whole implementation. However, key-components with an inefficient implementation can massively impact the performance of the whole system. A premier example is the location-transparent access to actors when sending messages. This essential primitive of the actor model should have minimal overhead in the local while supporting end-to-end actor communication thought the global Internet. Optimizing inter-actor communication in the network requires efficient handling of addressing and meta-data.

Processing overhead affects efficiency as much as memory management. Naturally, actors should exhibit a small memory footprint to allow running thousands or millions in parallel. When required, the runtime should make use of the available memory and scale out accordingly. Efficiency further concerns overhead for object access and interactions, e.g., when enqueuing messages into the mailbox of an actor or when interacting with handles of local and remote actors. With regard to the network stack, efficient organization is required for handling message content, sockets and potential buffers.

3.3 Flow and Congestion Control

Actors provide a scalable approach to processing large amounts of data. Since their communication is asynchronous by design, actors can send any amount of messages to other actors. An imbalance between average processing time and the rate in which new messages arrive can lead congestion and buffer overflows. Flow and congestion control address this problem. While algorithms to manage message flow and congestion are not a new topic in general and are often considered with regard to network protocols, a mapping and implementation for actor communication provides a few challenges. By design, actors do not have a feedback channel to communicate their workload to senders.

Relying on asynchronous message passing in concurrent as well as distributed settings, congestion and flow control are not only a problem for the network stack in CAF, but should be considered as a general addition to CAF actors. When using a transport protocol that is flow controlled such as TCP, the runtime might take information from the network layer into account when addressing communication with remote actors.

Challenges concern the creation of a feedback channel between actors, especially in scenarios that include multiple—or even changing—senders. Moreover, the reaction of an actor to flow control messages is not straight forward and strongly dependent on its behavior.

3.4 Identifiers

Unambiguous identifiers for actors and nodes which are valid among all participating nodes simplify network transparent communication. Knowledge of such an identifier should be enough to address an actor with a message or to spawn a new actor on a node. Identifiers are implementation-dependent and can be represented as human-readable names, randomly generated bytes or encode location information. Using an identifier to encode identity and location imposes a duality on the identifier as it determines not only who and actor is, but also where it is located which may impact scalability and mobility.

Mobility is not an essential characteristic of actors and only offered by some implementations. A major challenge for mobile actors is maintaining operation continuity during mobility handover. This requires mobility-transparent access to actors for both messaging and error propagation without message loss, unbound delays or buffering requirements. In general, moving an actor between nodes requires either a lookup service that keeps track of an identifier to location mapping—introducing a significant operational overhead to all interactions—or identifiers that are updated on all remote nodes—introducing challenges on how to handle messages sent during the update process. These challenges are confined to mobility support, which CAF does not offer.

3.5 Routing and Forwarding

Network transparent messaging between actors in a distributed system can be enabled in various ways. CAF builds an overlay between its nodes which (per default) establishes communication between nodes only when required. For the purpose of this discussion, a neighboring nodes can exchange messages directly, i.e., without the need to rely a message over another node in the overlay. For actors on neighboring nodes, network transparent messages passing is easily achieved as the runtime environments can exchange messages directly. Nodes can learn the identifiers of actors on non-neighboring nodes as actor identifiers can be shared freely in messages. The runtime environment is responsible for maintaining the transparency in these cases. Routing messages to the destination node can be handled in the overlay established by CAF or be left to the underlay network.

To enable routing in the overlay, each node could maintain a routing table and forward messages that do not address a local actor accordingly. While this introduces a dynamic topology, a node may act as a bottleneck if multiple nodes use it as a communication gateway. Management of routes in the presence of node failure introduces additional complexity. Providing ordering or delivery guarantees over intermediate nodes requires support from the runtime environment event if transport protocols already implement them.

Relying on routing in the underlay requires maintenance of the overlay topology. Nodes that reference actors of each other need to build a neighbor relationship. An advantage of underlay routing is that error handling and changes in the underlay topology are handled by the network. Moreover, standardized transport protocols deliver messages between neighbors and allow the runtime environment to rely on their guarantees, such as ordering or congestion control.

3.6 Scalable Communication

Communicating with peers in a distributed actor system requires a peer-wise state consisting of proxies for remote actors, state for message transport such as connections or addresses information, as well as failure detectors to monitor the liveliness or neighbors. Targeting scalability, state that scales with the number of actors should be avoided while state that scales with the number of peers should be considered carefully. An example is communication that requires message buffers to restore order or implement retransmits. While some state cannot be prevented (e.g., identifiers and sockets), uncontrolled peer-wise state can heavily impact scalability and thus performance. While providing upper bounds can limit state requirements, it requires handling associated failures such as full buffers. Working on the network layer, it should be considered how peer management can be implemented efficiently and how state can be minimized. In this context, the routing considerations from Section 3.5 should be mentioned. A reactive approach to building routing tables or neighbor relations can prevent the runtime environment from maintaining unused state. Managing the lifetime of distributed state is a challenge as it requires synchronization to propagate errors and prevent diverging state.

3.7 Message Transport

The current network layer of CAF is strongly coupled to TCP and inherits reliability guarantees for delivery and ordering for message passing. Moreover, the connection management of TCP is utilized to track the liveliness of connected nodes. In addition to guarantees, a transport protocol can provide deployment-specific adaption such as HTTP tunneling and WebRTC to enable connectivity in restricted environments.

Decoupling the runtime environment of CAF from a specific transport protocol widens its application domains and allows deployment-specific adaption. This is not only applicable to restricted environment, but addresses IoT scenarios where UDP and CoAP are important as well as performance driven system by enabling the use of InfiniBand. A challenge when exchanging the transport layer is decoupling the guarantees inherited from a specific protocol from the guarantees CAF provides for the communication between actors.

Throughout the Internet, communication security is addressed by transport layer additions, majorly TLS and DTLS. Opening CAF to an exchangeable transport layer eases the incorporation of secure messaging into the framework.

The guarantees for message transport in current actor systems often diverge between concurrent and distributed scenarios. For example, ordering of messages is often weakened from causal ordering in a concurrent scenario to FIFO or no ordering in distributed systems. Similar observations can be made for message delivery. This disparity impacts portability of applications as behavior might change with deployment.

3.8 Usability

CAF provides a high level of abstraction to developers while managing concurrency and distribution efficiently. Part of this abstraction is a high-level interface that reduces the complexity compared to the low-level interfaces of threading and networking. While simple tasks are easy to build with these low-level APIs, a scalable and efficient system depends on correct synchronization and careful resource management. The default configuration of CAF should enable developers to build scalable applications for concurrent and distributed systems. Developers that build larger or more complex systems should be able to configure CAF to fit their application. Adding new features to CAF should keep its setup complexity low and allow “out of the box” usage for developers that want to get started with CAF while exposing configuration options for advanced use cases.

In general, CAF aims to keep its core components dependency-free to allow building CAF with only a standard C++ compiler. Components with external dependencies can be enabled at compile time. Currently CAF offers an alternative networking backend that is based on Boost.Asio library [1] as well as the integration of OpenCL into the framework. Enabling these components at compile time keeps the overhead for standard uses slim while offering specialized functionality when desired.

[1]http://www.boost.org/doc/libs/ (Accessed October 2016)

4 Core Aspects & Related Work

Reliability, reachability, security and scalability are the core aspects of communication in distributed systems. These terms are often used but lack a clear definition. To define the objectives of communication in out actor system, each term is examined and defined in the context of this work.

This chapter starts with a brief overview over the components of CAF that are involved in communication. Thereafter, each section discusses a core aspect in detail, considering the solution space as well as the implications on the behavior of CAF. Related work is included in this discussion as well as the practices of other actor systems.

4.1 Components

Delivering a message to another actor involves several components which should be named to ease the discussion later on. Figure 4.1 shows an overview over a distributed system with two CAF nodes. Each node has multiple actors that exchange messages with actors on a local or remote node. Each node has its own runtime environment (RE) with a message passing layer, a middleman, a scheduler and an optional wrapper for GPGPU computations. REs communicate using the Binary Actor System Protocol (BASP) as an application layer protocol.

Behavior The behavior of an actor is implemented by users of the framework and defines how an actor processes messages. Actors can change their behavior at runtime. After an actor dequeues a message from its mailbox, it matches it against the handlers of its current behavior. A handler extracts the content of the message and offers it for processing.

Actor Each actor has a message buffer—called mailbox—which contains received messages that have not been processed. It can be accessed by the actor in order, sorted by arrival time or priority. Depending on the implementation, the mailbox size can be fixed or only limited by the available system memory. Furthermore, actors offer an interface to send messages to other actors via the `send` or `request` calls. Messages for local actors can be enqueued into the destination mailbox directly, while messages to remote actors are passed through the runtime environment.

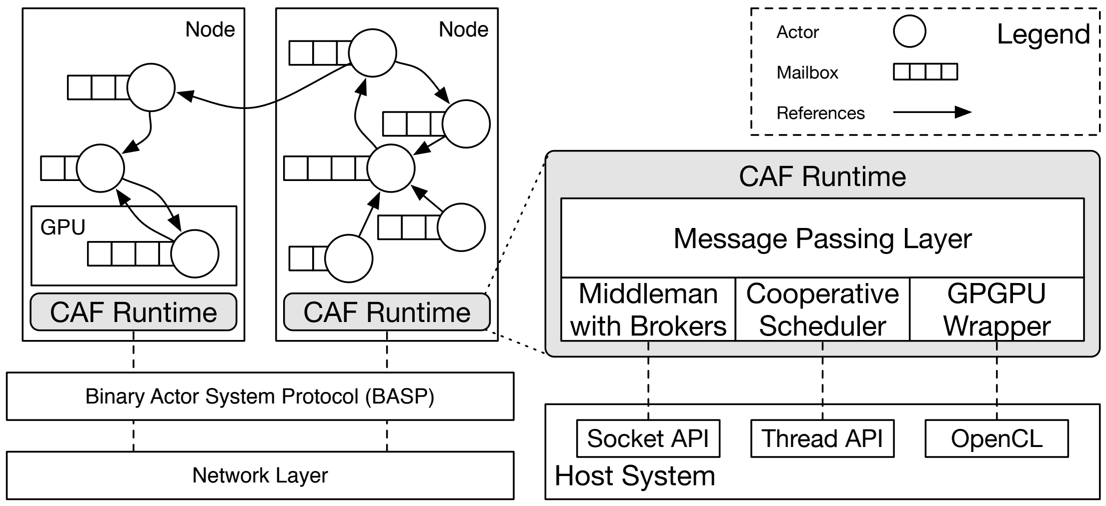

Figure 4.1: Architecture of an actor system build with CAF.

RE The runtime environment is an intermediate application layer between actors and the operating systems. It schedules actors and has a middleman (MM) that is responsible for communicating with other CAF nodes. The brokers of the MM manage network connections. They read chunks from the network buffer via the socket API and deserialize the raw bytes into messages. The RE may store messages temporarily before enqueueing them into the mailboxes of local actors.

Network Messages are transported from the application endpoint of the receiver over the network. Its interface is the protocol API of the operating system. Naturally, this component is not used for the communication between actors in the same RE.

Note, that the involved components differ for local and remote scenarios. Since actors can enqueue messages directly into the mailboxes of local actors, messages do not need to pass through the RE or be transported over the network. Although, the RE is still responsible for scheduling actors, it does not store or reference their messages.

4.2 Reliability

The design aspects of our protocol that relate to reliability are delivery guarantees for messages between actors, message ordering as well as monitoring remote actors. Each aspect has different levels of assurances that vary in strength as well as in operational overhead. In addition to laying out options and examining their characteristics, related work will be considered.

4.2.1 Message Delivery Guarantees

Message delivery specifies how a sent message reaches the intended destination. This section explores different delivery assurances and takes a look at messaging in different contexts such as other actor implementations. Providing reliability adds complexity and state to each actor-to-actor communication. Our goal is to provide guarantees that do not add much operational overhead to simple cases, but provide strong enough guarantees to fit the most common use cases. Developers should be able to rely on the default implementation, but have the opportunity to implement stronger guarantees for special cases.

As reliability is not a clearly defined characteristic in the context of message delivery in actor systems, we start off with a look at related work, before examining how messages are passed through the framework and what challenges we face to provide reliable delivery.

Related Work

Akka delivers messages unreliably with at-most-once semantics per default [21], i.e., a message is delivered either once or not at all to the destination mailbox. Included in the framework is a solution for at-least-once delivery in form of a persistence module which additionally allows actors to recover their state after a crash. Erlang is named as an inspiration for defaulting to weak delivery guarantees as it successfully uses a similar approach.

Armstrong defined message passing in Erlang "[. . .] to be unreliable with no guarantee of delivery" in his thesis [see 22, page 22]. The additional effort required to write applications that can handle unreliable message passing furthers scalability and increases robustness against errors. A later publication [5] goes into more detail on the topic and states that the reliability of message passing is dependent on the reliability of TCP. However, TCP itself is not enough to guarantee delivery to an actor. Errors in the RE can occur after a messages was accepted at the application endpoint, but before it was passed on. An example for this type of failure in a simple distributed Erlang setup is provided by Svensson et al. [23].

Microsoft released Orleans [24], an implementation of the actor model that targets clusters. It hides most of the distribution and error handling from developers. Failed actors are detected by the runtime and redeployed transparently before delivering a message. The RE favors availability over consistency when redeploying actors and accepts temporary inconsistencies such as actors performing redundant calculations. Per default, messages are exchanged with maybe delivery guarantees to avoid the associated costs in every message exchange. However, at-least-once delivery can be enabled, which retransmits messages until the receipt is acknowledged [1]. Since the runtime does not detect duplicates, using at-least-once delivery requires developers to handle duplicates in their implementation.

The article "Nobody Needs Reliable Messaging" [25] analyzes reliability in the context of SOA, Web Services and REST. It argues that reliability requires conformation on the application layer which makes an implementation on a lower layer redundant. A similar conclusion is drawn for duplicate message detection, e.g., a duplicate order in an online market would lead to the same messages with different sequence numbers on the transport layer. Related to this discussion, Saltzer at al. [26] explore the implications of end-to-end communication. Without knowledge of higher layers it might be tempting to provide more functionality than needed. While functionality can be implemented on top of communication systems, in some cases it may be beneficial to implement partial functionality on lower levels to enhance the overall performance. As a result, the assumption that avoiding redundancy improves performance should be viewed with care.

In his dissertation Agha argues the guarantee of communications delivery should be modeled as it eases the reasoning about the system [see 3, Section 2.4.1]. However, he notes that the buffers required for the communication are limited by nature which makes it impossible to ensure delivery in all cases.

The Message Delivery Process

Figure 4.2 depicts the process of sending a message to a remote actor. It shows two nodes, each with an instance of the CAF runtime environment (RE) which includes a middleman, an actor and a proxy of the remotely running actor.

A message addressed to a remote actor is passed to its local proxy. A proxy is not an actor, but transparently forwards the message to the middleman (MM) of the RE, which acts as the application endpoint towards the network transport. The MM uses brokers to manage connections between RE, serialize messages and send the packets. On the receiver side, the steps are similar, but the proxy is not involved. Packets are received by the brokers of the MM, deserialized into messages and enqueued into the mailbox of the receiving actor. Once the actor is scheduled and the message reaches the front of the mailbox queue, it is dequeued and processed.

(1) Local Operations Messages between local actors can be delivered directly and require no additional processing by the runtime environment. Local delivery may fail either due to

[1] `http://dotnet.github.io/orleans/Runtime-Implementation-Details/Messaging-Delivery-Guarantees` (Accessed February 2016)

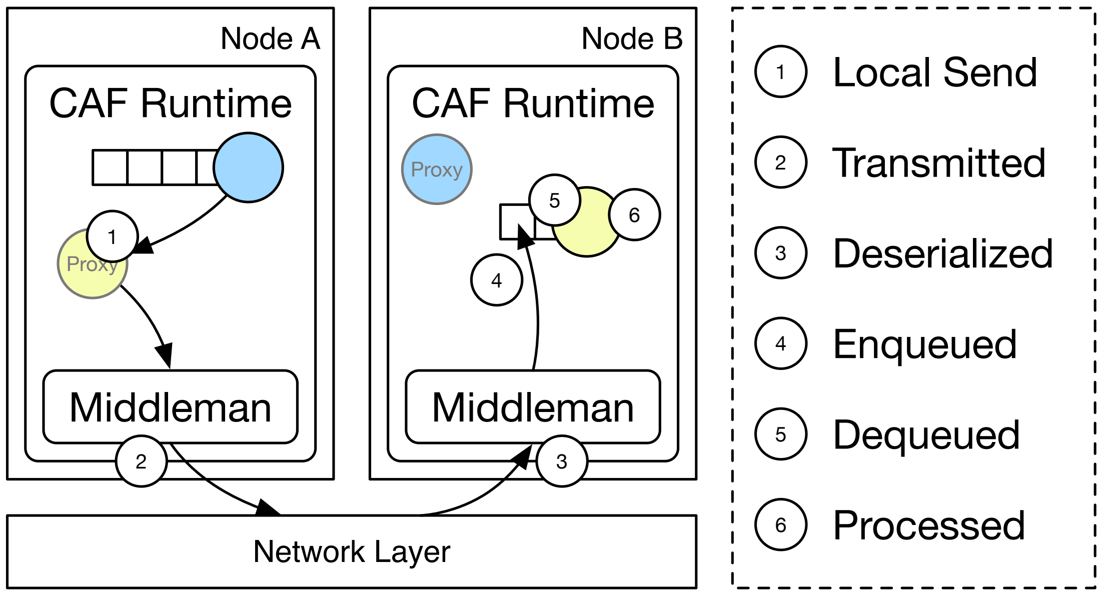

Figure 4.2: Message delivery between distributed actors.

limited memory availability when creating a new message or because of a bounded and full mailbox of the addressed actor.

In case the addressed actor is not local, the message is passed to a local proxy. Proxies do not have mailboxes, but transparently forward messages to the middleman. Like in the local delivery, an error occurs if no memory is available to create a new message. Bounded or full mailboxes, however, cannot be checked as the proxy does no have the required information.

To enable similar checks for remote as for local cases actors, could inform their proxies of the free space in their mailbox. Although this remote view is only an approximation and is subject to the delay of the network transfer, proxies could provide feedback earlier and provide similar errors for remote as for local messages.

Out of memory errors, on the other hand, are hard to foresee and heavily depend on the application code. Precautions include a slim framework design without memory leaks.

(2) Message Sent Sending the message is the next step. The application endpoint is the interface to the network transport, e.g., the socket API. To sent a message, the middleman has to identify the host of the receiving actor, serialize the message and sent the packets to the destination address. For this purpose, the middleman tracks handles for its connections to remote nodes and stores a mapping from host ids to these handles. When forwarding a message to the middleman, a proxy adds its host id to the message to identify the right handle.

Establishing a connection to a remote host and acquiring a proxy requires a call to the IO module of the framework. In addition to explicitly creating connections via published actors on remote nodes, the RE can learn of new actors via messages. Upon receiving an actor handle with an unknown host id in a message, the RE maps the new host id to the connection handle of the sender which can forward the message accordingly. Either way, once a proxy is created, the remote actor can be addressed with a message.

Attempting to look up the connection handle to a host that is no longer reachable or finding no mapping at all could be signaled using links and monitors, which are traditionally used to observe liveliness in actor systems. In addition, when loosing a connection because an intermediate node is no longer reachable, the RE could try to find a new route, via another node or through a direct connection, before signaling an error.

(3) Message Received Before a message is received, it is transported over the network, which bares several sources of errors. First of all, connectivity may be interrupted. For the sender, determining whether the problem originates from the network or from the receiver may not be possible. The actor model includes links and monitors to observe the liveliness of actors. Since this is a complex topic on its own, it will be discussed separately in Section 4.2.3. Next, even with continues connectivity, packets may be lost during transport or arrive more than once due to retransmits or errors. Although several transport protocols can provide some guarantees, we want to avoid relying on a specific one as we aim for an exchangeable transport layer. Finally, even if the packets arrive exactly once, local problems at the receiver side can prevent messages delivery. For example, full buffers in the network stack lead to message loss.

Many protocols offer ways to ensure packet delivery through retransmits based on timeouts with varying complexity. To guarantee delivery over the network, but stay independent of a specific transport protocol, we could use an application layer protocol that provides the desirable guarantees or implement a basic solution as part of the framework. In any case, the functionality should not enforce redundancy and operational overhead. In cases where connectivity cannot be recovered or delivery takes an exceedingly high amount of time, the RE should be able to notify the sender to satisfy the functionality of links and monitors.

Duplicate packet detection is also offered by some protocols. In addition, the runtime environment has the opportunity to detect duplicate messages after receiving messages but before enqueueing them into mailboxes.

(4) Message Enqueued After the packets reach the application endpoint, a broker in the middleman reads the buffer before deserializing it into a message. Limited memory may lead to a problem when the buffer is copied to user space. Additionally, to deserialize the buffer into a message the content has to have the right format and consist of known types or it is dropped. Besides memory limitations, actor mailboxes may be bounded to a fixed number of messages. Thus enqueueing a message into the receiver mailbox may fail.

The runtime environment has many opportunities to handle these challenges. Limited memory availability could be handled by buffering messages at the sender side and resending them regularly until enqueuing succeeds. This shifts part of the problem to the sender, which has to manage its own resources and cannot simply buffer messages indefinitely. Various flow control mechanisms offer more complex protocols that signal available resources to senders and enable the adjustment of the data flow as well as local computations to the capabilities of the receiver.

Problems that cannot be handled in a determinable time frame or at times not at all, such as unknown types, could be communicated back to the sender side and extend the existing capabilities of links and monitors. In many cases, detection is preferable to silent errors or endless retries. Even for memory errors, these messages could be preallocated to enable sending them with a copy operation to the network layer.

Enqueueing a message into the receiver mailbox, is the last step with an assessable duration and should not take much longer than the transport over the network. However, the next time the message is handled depends on scheduling, the processing time of previous messages and the behavior of the receiving actor.

(5) Message Dequeued Dequeueing a message is the last step before handing control to the application logic. This happens once the receiving actor is scheduled and it has processed all messages that arrived previously or have a higher priority. Since the processing time is implementation dependent and not limited, this may happen at any point in time after the message has arrived or, in the worst case, not at all. Consequently, the sender cannot distinguish between a delayed and a lost message.

After dequeueing a message, the actor matches it against its behavior and processes it according to the matching message handler. If no handler matches, the executed action depends on the default handler of the actor: either it skips the message, i.e., moves it to the back of its mailbox, or simply drops it.

While the dropped messages could be communicated back to the sender, handling long delays until a message is dequeued requires more effort. Defaults or timeouts based on network characteristics are not a viable estimate here as the time a message spends in the mailbox varies greatly for various use cases. User-defined timeouts are one possibility to eventually force errors—presuming that developers have enough information to provide valid estimates. These timeouts do, however, not prevent dequeueing and processing of the original message at a later point in time.

(6) Message Processed Acknowledging message processing crosses the line from the framework implementation to the application logic. When considering end-to-end communication, knowing that a message was processed is more valuable than knowing that it was delivered [26] as delivery does not guarantee processing.

Estimating the time until a message has been processed faces similar challenges to estimating the time an actor starts to process it. Once again, user-defined timeouts are an approach to force an error at some point or simply when processing of a message is required within a definable interval. Cases where eventual processing is of importance could be split into two steps and shift part of the responsibility away from the sender. In a first step the message is delivered to the mailbox of the receiver, which is ensured by the sender. The second step is handled locally by the receiving RE and ensures that the message is processed eventually. The sender would still be required to observe liveliness of the receiver between delivery and processing to signal errors to local actors.

Another challenge occurs when an actor forwards—or delegates—a message. Since this happens during its behavior the message would be handled by an actor, but not processed completely. This breaks end-to-end message passing between actors and prevents the sender from tracking the message further as it might not even know the new receiver. To handle such cases, the delegator could be used as an intermediate node for communication or even assume responsibility for further delivery.

Discussion

Providing reliable transfer including all the steps in Figure 4.2 results in six steps. We examine the gain and costs to guarantee delivery for each one. The baseline is local delivery which works reliably. As CAF does currently not support bounded mailboxes, this step will fail if no memory is available to create a new message. Memory management of the overall application, however, is a responsibility of the user and cannot be handled by the framework besides providing a slim implementation without memory leaks.

There are several ways to handle a bounded and full mailbox. Messages could be buffered at the sender and be resent after a short timeout, which is easy to implement for local cases. While this is not a problem if it happens on rare occasions, buffering many messages at many senders which target a single source can get out of hand easily. Especially if the processing time is longer than the timeout, this behavior does not help to reduce the overload. Next, the failure to enqueue a message into a mailbox can be answered with an error messages that are dropped by default, but can be trapped and handled like error messages from links and monitors. To prevent the need to buffer messages at the sender, the original message could be included in the error. While this does not entail additional copy operations, passing these messages around introduces another form of buffering. Finally, flow control mechanisms prevent buffer overflows with different strategies, generally implementing a feedback channel between sender and receiver to agree on transmission rate.

Failures to send messages due to unreachable destinations are caught and propagated by links and monitors. Before raising errors the runtime environment may try to fix the error itself. Messages sent during this recovery process need to be buffered locally. In case the failure cannot be corrected, the buffered messages are either lost as well or require an additional recovery process. Raising an error early reduces the amount of buffered messages and minimizes the necessary recovery.

Message delivery on the network layer is a significant step as the network is one of the more likely sources of errors. A transport protocol that acknowledges delivery such as TCP only propagates failures for delivery to the network buffers. Thus, it falls short of end-to-end transport. In contrast, acknowledging delivery on the application layer informs the runtime environment exactly which messages were delivered and acknowledges that the delivery did not fail due to memory limitations as the message was already copied from the network buffer into user space. In the case of CAF, a received and acknowledged message would be a deserialized BASP message.

Enqueuing messages into remote mailboxes faces similar challenges as delivering to local mailboxes, i.e., available memory and mailbox space. In case delivery was not acknowledged on the application layer, additional steps are required to copy the message from the network buffer to the user space and deserialize the message thereafter, both of which may potentially fail. This is the first step that ensures receipt by the runtime independent of the acknowledgement mechanism of the network transport. While processing is not ensured, this is guaranteeing delivery up to this step is the remote equivalent of a local send operation.

After a message is enqueue into a mailbox, its dequeueing does not provide much additional information. In the worst case, processing the message leads to failure of the processing actor or it only enqueues the message in its own or another mailbox. The absence of a time frame for this operation to happen results in an unattractive point for a delivery guarantee. Applications that require this information can achieve similar behavior by sending a message as the first operation of a message handler.

Acknowledging message processing provides the most interesting information considering end-to-end message passing. At the same time, addressing a generalized use-case is a very complex task, as the application logic dictates much of the necessary context. Processing time per message, average delay in the mailbox, current load and the messaging interface of the receiver all influence whether a message is processed and how long it takes to process the message after if was received. As a result, a reasonable failure case cannot be defined for all scenarios.

4.2.2 Message Ordering

All communication in actor systems works by exchanging asynchronous messages. Message ordering describes relations among messages exchanged in the system. Since we don't want to require a transport protocol that maintains order, either an application layer protocol or the runtime environment can restore the order of received messages. In this context, the order only determines how messages are enqueued into mailboxes as it may be changed afterwards. Mailboxes could sort messages by priority or actors could process messages out-of-order. There are four orderings with increasingly strong assurances that we consider here: *non-deterministic*, *first in - first out*, *causal* and *total*.

Without the order requirements, messages can be received in a different order than they were sent in. While the order may be the same, the system does not take any action to restore or enforce order after it was broken.

First in, first out ordering (FIFO) means that messages that are sent first arrive first. This guarantee only creates a relation between messages from a single sender and is not transitive. Transitivity would maintain order even if a message is received and forwarded by an intermediate node. Figure 4.3 shows an example for non-transitive message ordering as message c is received before message a. Before enqueueing messages into mailboxes, the RE could buffer them for a short time to restore order determined by sequence numbers in the messages. Configuration parameters include the time frame during which messages are buffered and out-of-order messages can arrive before they are considered lost as well as buffer sizes.

The "happens before" relation [27, 28] describes the logic of causal message ordering. Unrelated messages are determined to be "concurrent" or "independent". Hence, causal ordering is not restricted to messages exchanged by a pair of actors, but can establish a relationship between messages in the whole system. The message exchange in Figure 4.3 breaks causal ordering as message a happened before messages b and c and should thus be delivered previously. In this case, the order could be restored by buffering message c until the earlier message a is received. There are various algorithms to establish a causal message order in a distributed

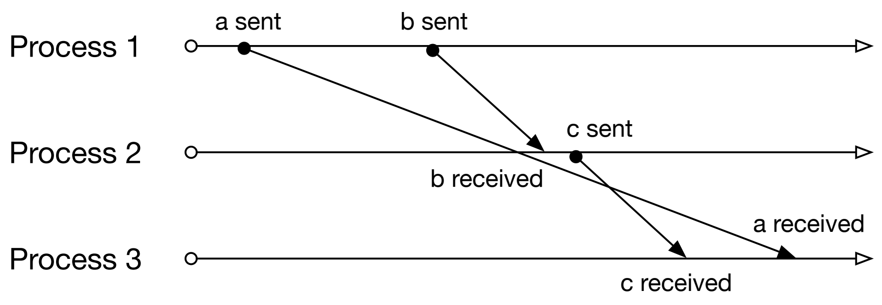

Figure 4.3: FIFO ordered messaging that violate causal order.

system, e.g., the Birman-Schiper-Stephenson Protocol [29] or the Schiper-Eggli-Sandoz Protocol [30]. While both algorithms use a form of vector timestamps to determine if causally preceding messages have been delivered, the first one additionally requires messages to be exchanged via broadcast. Note that an event e_1 which "happens before" another event e_2 does not necessarily have caused the latter, but both are still in a relationship as per causal ordering.

A total order extends causal order and gives order to all messages in the system and not only for the causally linked. Hence, all messages arrive in the same order at all receivers. Introducing a total order requires the synchronization of all participants. To achieve this, the totem protocol [31] passes a token around in a logical ring, which allows the owner to broadcast messages. Until the token is acquired, messages are buffered locally. An alternative approach could be a central sequencer that provides sequence numbers for all messages and advances the time.

Related Work

The actor system Orleans [24] is an example of a framework that does not enforce ordering. It wants to avoid the related impact on scalability as well as the overhead in processing power and state that is required to restore the order of received messages. CAF follows a similar approach and currently does not maintain the order of messages actively.

Erlang and Akka both enforce FIFO ordering. Although Erlang defines this ordering as part of their basic rules of message passing [see 22, page 25], the decision is not further explained besides stating that it eases application development. Akka stresses that this is only true for the order in which messages are enqueued into the mailbox [21] as the mailbox itself may change the order, e.g., by prioritizing certain messages. In particular, system messages such as errors use special mailboxes and may be delivered out-of-order. Akka implements ordering on top of TCP [2], but utilizes additional per-connection queues to sort messages and handle errors such as TCP reconnects and full buffers.

Long et al. [32] explore what causes ordering problems in message passing systems. The three main semantics they identifies are (1) *synchronization*, split into asynchronous and synchronous messaging, (2) *processing*, i.e., non-deterministic vs. in-order delivery and processing, as well as (3) *sharing* in the form of data sharing or data isolation. For example, code that looks sequential but depends on asynchronous unordered messages may lead to undefined behavior. They build message passing model by combining different aspects of these semantics. Their base model uses asynchronous message passing, with non-deterministic message delivery and processing as well as data sharing semantics. The other model are build on base by exchanging different aspects as well as adding transitive in-order delivery. A static analysis is used to evaluate how programs are affected by ordering problems when exchanging messages with these models. For framework designers, they see in-order delivery and data isolation as the most critical semantics.

The Pony language ensured causal message ordering in an earlier approach to distribution. In addition to easier debugging in and reasoning about distributed systems, the garbage collection of the language depended on the this property. An informal view as well as a formal argument were discussed [see 33, Chapter 4]. The idea is based on organizing nodes in a tree structure and route messages along the (unique) shortest path from one node to another. Combined with a TCP-based ordered message exchange between nodes, messages are guaranteed to arrive in causal order.

Ordering for Messages in Actor Systems

Easiest to implement is non-deterministic ordering, which leaves developers with FIFO ordering for local delivery and transport layer dependent guarantees for remote delivery.

FIFO ordering can be implemented with different granularities. It requires a consecutive sequence number to determine order as well as a buffer to restore it. Implementing both for each actor pair requires a lot of state for buffers and to track sequence numbers. Moreover, cleaning up buffers when actors exit requires synchronization. These requirements could be reduced by sorting messages not on an actor to actor basis, but per connected node. Although this applies order to potentially unordered messages, the management is no longer part of each actor and can be handled by the runtime environment. In the presence of unreliable messaging

[2] `https://groups.google.com/forum/#!topic/akka-user/w2iFDpxiF6U` (Accessed February 2016)

a timeout or maximum buffer size is required to deliver messages even if one is lost. These constrains may delay all messages and need to be carefully considered.

An overview and categorization of algorithms that provide causal order can be found in [34]. The author identifies two categories among generalized algorithms, those that have non-causal latency and those have no non-causal latency. The trade-off between these categories is delay in the former in contrast to increased message sizes in the latter. Delay is created by synchronizing processes, e.g., by using synchronous messages or exchanging Lamport timestamps. The lower bound for the increase in message size can be calculated. The additional information that need to be exchanged are time vectors in the system with a size equal to the number of processes n [35]. Moreover, to determine causal dependencies for transitive message passing with more than one intermediate node at least n vectors are necessary [36], aggregated to time matrices by some algorithms. These constrains can be weakened for special use cases such as adhering to fixed routing topologies such as rings or a tree topology as shown by the Pony language. Still, in the worst case messages are routed from one leaf through the root to another leaf which introduces latency. In addition the topology has to be maintained when nodes join, leave or fail.

Implementation of a total order is straight forward. One node in the system is chosen as a sequencer which receives the messages from all actors and broadcasts them in FIFO order. Such an approach introduces a strong coupling in the system as even local messages would have to pass through a possibly remote sequencer. This impacts scalability greatly.

Discussion

Local delivery in CAF leads to a causal ordering among messages enqueued into a mailbox. This is a result of implementing mailboxes as lock-free FIFO queues which are accessed by actors in a single non-blocking but synchronous call when sending messages. Although the current implementation for remote messaging is strongly coupled with TCP, some cases allow messages to arrive out-of-order. For example, nodes that are not connected directly can exchange messages via different intermediates which breaks the ordering guarantees of TCP. In addition, errors such as dropped messages due to full buffers or TCP reconnects break this assumption. Thus, developers cannot rely on it, i.e., there are no ordering guarantees for messages sent to remote actors.

The main advantage of a defined message order is an easier development and testing process. When messages that are sent by sequential statements will be delivered in the same order, reading code and considering side effects gets easier. Relying on the same ordering for local as for remote messages prevents deployment specific bugs and eases porting local applications to distributed systems. In the same way, reproducing failures is easier to achieve if communication is ordered and predictable.

Order is not established for free and introduces overhead such as delays, increased message sizes or numbers, as well as required buffers. With stronger ordering guarantees the coupling between actors and nodes is also strengthened. These constrains may impact performance and scalability of the system, both of which are highly undesirable.

Restoring FIFO order among received messages comes at the cost of buffers and a potential delay. The maximum delay until a message m is delivered is n times the timeout until a message is considered lost where n is the number of messages in front of m. This should be considered when choosing the buffer size and the timeout. Ordering cannot be performed in a single buffer, but requires one buffer per peer runtime environment.

Implementing causal order comes at significant costs. Relying on synchronous communication introduces a strong coupling between actors and nodes. While synchronization on a local machine may be fast compared to synchronization with remote machines, the framework is highly optimized for efficient message passing and scheduling. Alternatively, adding vector timestamps to messages greatly increases the amount of data exchanged in the system. In addition, hosts can schedule high amounts of actors, frequently spawning new ones that only run for a limited time or task. A changing amount of participants is generally not handled well by vector clocks. Neither relying on synchronous communication nor on broad- or multicast is an option for the default communication in an actor system.

Total order is not a desirable property for the messages exchanged between actors. By definition, actors are concurrent and isolated entities. Adding such a strong coupling, e.g., in form of a central sequencer, to all their communication impacts scalability and performance. While some use cases may justify the overhead to maintain a total order, the majority of cases does not. As such it is not a good candidate for the default ordering.

While ordering eases software development, strong ordering guarantees are costly and introduce the need for synchronization. FIFO ordering has a comparably low overhead and provides part of the ordering characteristics of local messaging to remote messaging. As such it is a tradeoff between desirable properties and overhead.

4.2.3 Reliable Monitoring of Remote Actors

Components in a distributed system can fail independently which leads to partial failure of the system [37]. There is no central instance such as an operating system that can reliably detect failures. Detecting failures and propagating knowledge about them through the system are key aspects when building reliable and robust systems. Resilient systems additionally have to recover from failures by correcting errors such as an inconsistent state among nodes.

The inability to distinguish between the failure of a remote node, a link failure and a very slow link or node adds uncertainty to detecting crashed nodes. However, failure detection is a crucial component for some algorithms such as reaching consensus. As shown by Fisher et al. [38], this is not possible in a distributed asynchronous system if participants can crash without reliable detection. This uncertainty is addressed by Chandra et al. [39] with the introduction of unreliable failure detectors. The unreliability characteristic allows failure detectors to make mistakes such as wrongfully suspecting a process to have crashed. In addition to a lack of reliability, these detectors must satisfy the properties *completeness* and *accuracy*. Completeness states that every failure is detected eventually while accuracy means that a detector eventually drops wrongful suspicions, i.e., the failure detector converges to correctness. Since eventuality is hard to pinpoint, practical applications usually rely on a trade-off between fast detection times and a higher probability that a failure is correctly detected. In conjunction with failure detection the term "suspect" is often used to emphasize the uncertainty.

Bertier et al. [40] define two basic strategies for detecting failure of remote nodes, heartbeats and "pinging". Heartbeats are sent regularly by each monitored node to its peers, implementing a push approach. Each peer then tracks the arrival of such heartbeats to determine reachability. The speed and reliability of this strategy are influenced by the interval between heartbeats as well as the timeout or accepted number of lost messages before a peer suspects it. In contrast, "pinging" is a pull approach that regularly requests a response. How often a request is sent or how soon an answer is expected determine what is necessary to suspect a peer to be unreachable. The advantage of these approaches is a relatively short detection time at the cost of network load and processing power as caused by regular message exchanges.

Traditionally, failure detectors return a binary value that indicates the connectivity or liveliness of the supervised entity. This value is calculated based on configuration parameters such as heartbeat intervals or a tolerated number of lost messages. However, developers can only bind actions to the state change when the connection has already been lost. In contrast, an accrual failure detector returns a (positive) suspicion value on a continuous scale and leaves the interpretation to the developer. The higher the returned value, the more likely it is that either the connection or the supervised node have failed. When plotted over time, the suspicion value returned by an accrual failure detector has to satisfy four properties [41]:

(1) *Asymptotic completeness* states that on an infinite timeline the suspicion value of a faulty entity tends towards infinity. (2) According to *eventual monotony* there is a point in time after which the suspicion value of a faulty entity is rising monotonically. (3) A monitored entity

works only correct if an *upper bound* for its suspicion value exists for an infinitely long run. (4) The suspicion value of a correct entity always returns to 0, i.e., it always *resets*.

Actions can be bound to different output values such as preparing for probable failure before taking more decisive actions once a node has failed. Alternatively, the output can be used directly, e.g., to compare the trust in workers and assign tasks accordingly.

Related Work

Chen et al. [42] discuss quality of service (QoS) measurements for failure detectors and introduce a heartbeat based failure detector that can be customized to meet different QoS requirements. The configuration parameters are: 1) the heartbeat period, i.e., interval between sending heartbeats, and 2) the timeout delay, i.e., the time after the receipt of a heartbeat until the sender is suspected. If a heartbeat is received from a suspect, the suspicions are dropped. Instead of calculating the arrival time of the next heartbeat by adding the timeout delay to the most recent arrival time, a sequence of expected arrival times is calculated in advanced, based on the time σ_i the message m_i is sent and adjusted by the timeout delay δ. This sequence is called "freshness points" because only fresh heartbeats are accepted after such a point is reached, while heartbeats that arrive late are ignored. This increases the accuracy of the detector as heartbeats that do not arrive on time do not influence the expectations on future arrival times. Chen et al. further show how to calculate the configuration parameters to adjust to quality of service requirements. To adjust to changing network connections, the author suggest a dynamic recalculation of the parameters.

The sending time σ and the timeout delay δ are not reliable estimates in the presence of unsynchronized clocks and unknown messaging behavior, which are both common. A variation of the basic algorithm above estimates the arrival time by tracking the deviation between the sending and the arrival time of recent heartbeats (as seen on the local clock) and adding the average to sending time σ as basic estimate. To account for the uncertainty, a safety margin α replaces δ. It can be calculated to meet a targeted upper bound on detection time.

Based on this model, Bertier et al. [40] propose an extended algorithm. In addition to restructuring the arrival estimate to a recursive calculation, the calculation for the safety margin α is adapted to use Jacobson's estimation of the TCP delay for retransmits. An additional second layer is introduced to adapt the first layer, i.e., the failure detector, to different applications, for example, by tracking statistics or collaborating with remote nodes.

Erlang introduced monitors and links to build fault-tolerant systems. A monitor is a one-way connection between a pair of actors, where the monitoring actor is notified in case the monitored actor fails. Monitors send `DOWN` messages which are dropped by the receiver unless its behavior explicitly provides a handle for them. Links provide a bidirectional connection and, per default, crash the receiver of the `EXIT` message with the same reason. Links and monitors are designed to work in local as well as remote cases. When observing remote actors, failures are not limited to crashes of actors, but include link and node failure. The detection of failures and liveliness in Erlang is discussed in [23]. The failures of remote nodes are detected using heartbeats, also referred to as ticks [3]. A node is considered down when a configurable number of consecutive ticks sent by it do not arrive.

Akka uses heartbeats as well [21], but implements an accrual failure detector. Specifically, Akka uses the "ϕ Accrual Failure Detector" by Hayashibara et al. [41], which aims to be flexible and adapt to shifting network conditions. The failure detector stores the arrival times of recent heartbeats to extrapolate the probability that the next heartbeat message arrives in the future, i.e., that it arrives more than t time units after the last heartbeat where t is the interval between the receipt of the last heartbeat and the current point in time. The suspicion level ϕ is then calculated as the negative logarithm of the probability that the heartbeat arrives at the current point in time.

For the evaluation, Hayashibara et al. compare the ϕ failure detector to the failure detectors of Chen et al. [42] and Bertier et al. [40]. The metrics for this evaluation are the average mistake rate and the average detection time as proposed by Chen et al. in [42]. Data for the evaluation was collected from nearly six billion messages sent over an transcontinental link via UDP. In the scenario, the ϕ failure detector showed its strengths for fast detection time where it has a lower mistake rate than the other detectors. However, it was outperformed by the Chen failure detector for longer detection times. Still both failure detectors provide similar and much better results than the Bertier failure detector.

A focus on minimizing resource consumption and messaging overhead leads to lazy failure detectors. Fetzer et al. propose a strategy that requires every regular messages to be acknowledged [43]. A monitored process can than be suspected depending on pending message receipts and previous round-trip times. Satzger et al. aim to reduce the amount of explicit heartbeat messages further [44]. Their detector only sends heartbeats if no message was exchanged within a certain interval and thus avoids regular heartbeats as well as acknowledgments during active communication. Although this approach increases the messages size because it depends on piggy-bagging sequence numbers and a time stamp in regular communication, it can largely reduce the number of total messages exchanged.

[3]`http://erlang.org/doc/man/kernel_app.html`, see `net_ticktime` (Accessed February 2016)

Failure Detection for CAF

Following the concepts introduces by Erlang, failure propagation and its user interface are based on links and monitors. The runtime environment propagates errors from local actors that exit with an abnormal exit reason. The reliability for delivering these messages to remote actors depends on the delivery guarantees discussed in Section 4.2.1. To detect link and node failures, CAF depends on the strong coupling to TCP. It observes TCP connections[4] and propagates their failures in the local system. Regular heartbeats can be enabled to detect failures faster while no other traffic is present.

Handling the detection of remote and local failures separately is necessary as both manifest very differently. The failure detectors discussed here only concern remote failures. To allow for an exchangeable transport layer, the coupling to TCP has to fit the transport and application layer protocols in use. To offer reasonable defaults, CAF could offer exchangeable network stack modules that bundle compositions of suitable components.

A properly defined API eases the implementation of these modules and gives developers the opportunity to optimize the network stack for their applications. Considering the failure detectors discussed here, an implementation requires the ability to send error messages and heartbeats, to monitor incoming heartbeats and in case of a lazy approach append additional information to exiting messages.

Discussion

Failure detection and propagation is an integral part of the actor model. There are several components in an actor system that can fail: actors, runtime environments (REs), nodes, and network links. As discussed, local and remote detection requires different mechanisms.

For actors however, deployment should be transparent. Actors can move away from the host they were spawned on and are not bound to the lifetime of a fixed node. While node failure may be of interest to implement resilient systems and recover from failures, actors should not need to know the mapping between hosts and actors. Moreover, to keep the deployment transparent to the application developer, no new error types should be introduced. This requires the new failure type to be mapped to deployment-independent errors, i.e., to propagate node and link failure as the failures of individual actors hosted on the crashed node.

The differences between deployment in concurrent and distributed settings cannot be completely hidden from developers. APIs often require specific information such as host and port, for example when moving actors between nodes or spawning new ones remotely. Failures

[4]The TCP timeout differs by implementation. RFC 1122 recommends it to be at least 100 seconds [see 45, pp. 101].

associated with these actions have to carry the relevant information to be meaningful. A trade-off could be to throw distribution specific errors only when explicitly requested and offer a default mapping while the RE attempts to recover.

The links and monitors implemented in CAF map the binary output of traditional failure detectors to the error messages. This binary interaction cannot make use of all capabilities of an accrual failure detector, i.e., it could only react to a single configurable threshold. Extending failure messages to allow for multiple thresholds leads to different failure models for local and remote scenarios, where some messages could only be received from remote actors. In contrast, the traditional messages (`DOWN` and `EXIT`) are received independent of an actors deployment.

4.2.4 Discussing Reliability Guarantees for CAF

Actor frameworks like Akka and Erlang encourage developers to write applications with distribution in mind. Considering the related error cases during development allows transparent deployment in local as well as distributed systems.

While the actor model addresses concurrent as well as distributed scenarios, applications build on top of it often require adjustments to move from a concurrent to a distributed system. However, it might be favorable to allow all applications built built with an actors system to be deployed in both contexts transparently. While this may include overhead in local cases, it allows applications to scale much higher when deployed in a distributed system without the need to rewrite parts of the application.

The guarantees made to developers are usually constrained by remote deployments. This ensures that applications developed with these limitations in mind can run independent of their deployment. An example is the documentation of Akka, which does specifically mentions the differences, but encourages developers to develop applications that work in distributed deployments—which usually requires extra work.

This section discussed various guarantees for delivery, ordering and failure detection and examined the costs to provide the same guarantees in concurrent and distributed environments. In a local concurrent context CAF provides reliable message passing—provided the node does not fail—causal message ordering for messages enqueued into the mailboxes of actors and reliable failure propagation ensured by the runtime environment. These guarantees hold for a CAF node running actors concurrently in a process. Moving to distribution, CAF relies on the TCP to provide guarantees for communication with neighbors. Since TCP failures are not handled by the framework the limitations of the transport protocol affect the messaging of CAF as well. Moreover, the ability to route messages over multiple CAF nodes weakens the guarantees even further as changes in the routing topology and failures on intermediate nodes weaken the messaging guarantees even further.

Moving forward, CAF should offer the option for reliable message delivery, but provide no guarantees per default—falling back to the guarantees of the transport protocol. Making reliable delivery a default adds overhead to all deployments, which might be often undesirable. Still, enough use-case require delivery guarantees to warrant an implementation as part of the framework. Regarding ordering, the guarantees for distributed communication are weaker compared to local communication. Since causal ordering comes at a high cost, the default for actor messages will be FIFO ordering. While relatively cheap, it provides a significantly improvement over no ordering and eases reasoning about the systems behavior. Lastly, error propagation exhibits the greatest differences between concurrency and distribution. For simple applications, a mapping of node failures to their hosted actors may be enough to enable viable error handling. However, more complex application depend on the ability to handle failures that stem from distribution and require addressing network or node failures explicitly to implement proper error handling.

4.3 Rendezvous and Reachability

Building distributed actor systems involves the management of participating nodes to allow actors to interact with remote actors. Most notably, actors need to learn about the existence of other actors and, in a second step, be able to establish contact.

While some developers might have enough knowledge about their applications to implement statically configured rendezvous for their actors, the deployment environment is often unknown during development. As can be seen when introducing new IoT devices into home environments or when allowing elastic cloud services to scale over many nodes on demand. A rendezvous mechanism describes how the meeting process works in a defined context. For this purpose, shared knowledge such as a known rendezvous point, a multicast address or a name may be required.

A known name can be used to look up contact information using a *name service*. This requires knowledge how to contact the name service itself. Running a central instance ensures that the most recent information is always acquired at the cost of robustness. Distributed name services can be joined in a *federation*. This introduces synchronization requirements to exchange information and to redirect requests to the responsible instance. The increase in robustness introduces complexity to provide consistent information, detect failures and recover from them. The domain name service (DNS) [46] is an example of a federated name service. Names are translated to locators by delegation of the query to hierarchically structured name servers.

Data can be delivered through a network to a denoted location by *routing* it accordingly. This requires binding addresses to nodes in the system to specify the destination and determine the routing path. Ideally, addresses are organized in a way that allows efficient routing. Mobility poses a challenge for routing as the location of addresses change dynamically. Either changes to routing paths need to be propagated accordingly or the address binding is changed to reflect the current location. Commonly deployed networks are based on IP and thus offer routing based on IP addresses. Address can configured statically or be assigned dynamically, e.g., using the dynamic host configuration protocol (DHCP) or the IPv6 autoconfiguration. Mobile IP introduces a location-independent and stable home address in addition to the changing and temporary care-of-address of the mobile node.

Names can be overloaded with location information which can then be used for routing or simply used for routing directly. An effort to research name-based routing is done in the context of information-centric networking (ICN) [47].

An *overlay* network builds a logical network on top of an existing network using a subset of the existing nodes. The new topology does not necessarily reflect the physical deployment. The original network is still used for connectivity and routing. As such, a direct hop in the overlay can consist of multiple links in the underlay. A common use case is the organization of data in a decentralized system, for example in a distributed hash table (DHT). This data structure distributes the space covered by a fitting hash function among its participants and quick routes lookup requests to the responsible node through the overlay. Peer-to-peer systems often deploy DHTs due to their resiliency and scalability.

As with nodes, actors need to be identifiable unambiguously in the system to exchange network transparent messages. Depending on the implementation, names are assigned to actors by the runtime environment or by developers when an actor is spawned. The lifetime of actors is not necessarily bound to their initial node as some frameworks can move actors between nodes at runtime. This change in context should not affect their communication and requires considerations regarding names and reachability.

Relying on location-independent names, i.e., names that carry on information or identifier related to their location in their names, requires consensus among nodes to prevent name collisions. Since nodes may join the system in the future this might not be possible, but at least impacts performance and scalability.

It should be considered if names carry location-dependent information and if these information are used for routing purposes. In case names are solely used to provide identity, including a location-dependent part in the name is an easy way to prevent name collisions as the generated names only need to be unique within the context of their original host—provided the node is able to generate a unique identifier. Migrating nodes named this way only invalidates their location and not their identifier.

Should the location-dependent part be used for routing, its identity is overloaded with location information. As an example, each actor name could be prefixed with the host name of the node it is deployed on. While this might simplify message passing and the propagation of actors because the name alone is sufficient to locate it, actor migration is complicated if the individual part is not unique in the system. Migrating such an actor does not only invalidate the location information, but also invalidates its name.

4.3.1 Related Work

Erlang utilizes location-dependent names for nodes and processes. A complete node name includes the hostname and can be used to identify applications in the network and be used for routing. In the same way, locally registered process names can be used remotely in combination with a complete node name. The Erlang port mapper daemon (EPMD) is responsible for managing node names and runs as a separate process on each host. The daemon acts as a name server for applications and is tasked with tracking connections and names in the system. When nodes connect, the local daemons proactively exchange contact information and, per default, build a full mesh network between all known nodes. Once connected, processes can be spawned on the remote node or RPC can be used to interact.

In a similar fashion, Akka encodes location into names, differentiating between actor references and actor paths. The first one identifies a specific actor instance and is only valid as long as the actor is alive. In contrast, an actor path is a logical structure build from the name of the actor system followed by the supervision hierarchy of the denoted actor, starting at the system guardian actor and following the child nodes towards the actor. A path does not depend on the existence of the actor and thus remains valid if an actor at the location dies or a new one takes its place. Both can be either purely local or valid in a remote context. The latter variants contain information about the transport protocol in use, the name of the host actors system as well as host and port information. A path can be resolved to a reference by sending an actor at the location an `Identity` message though the local actor system.

SALSA Lite [10] assigns names to actors to identify them in a distributed system. It differentiates between purely local actors, actors that are addressable remotely and actors that can move between SALSA nodes [48]. Mobile actors require registration at a name server that can subsequently be used to acquire actor references by name. The name server itself is a non-mobile remote actor that can be identified via its name. Remote actors can be published at a specified port at the host under a unique name to allow lookup from remote node using their these information. Separating actor types according to their functionality allows optimizing for their use case by keeping the overhead of remote operations separate from purely local actors. Once actors are created sending them messages works transparently without knowledge what kind of actor is addressed.

4.3.2 Managing Distribution in CAF

Each CAF runtime environment (RE) generates a unique node id on startup to distinguish nodes in a distributed system and on a local machines [5]. Actors are assigned an increasing integer value by the RE which is unambiguous when combined with the node id and does not requires synchronization for its generation. Neither id encodes location information such as IP addresses and thus requires an additional mechanism to be resolved to a location.

There are two ways to connect to remote runtime environments in CAF, creating explicit connections and reactively learning about new nodes when their ids are received. The first method requires explicit calls on both systems and can be used to bootstrap a connection to a remote actor system. Using the function `publish`, a runtime environment listens on a specified local port for connection attempts to a given local actor. Remote nodes can create a proxy of the published actor by using the function `remote_actor`, passing host and port as arguments. Node ids are exchanged during a handshake and used for sending messages to the right node.

Once a connection is established, references to actors can be exchanged in messages which implicitly leads to the creation of new proxies. If an actor with an unknown node id is received in a message this way, the RE creates a new entry in its own routing table to track which node can forward messages to the new node. Alternatively, the RE can be configured to establish a direct connection to newly encountered nodes. For this purpose, nodes share contact information in form of host and port when asked.

The rendezvous mechanisms currently deployed in CAF is based on reactive accumulation of knowledge. Given the initial information to connect to a remote node, CAF learns the names of remote nodes only after the connection is established. In the same way, encountering new actors results in new proxies and new routing entires in RE given their host nodes were previously unknown. While this approach is not as resilient as a full mesh, it requires little overhead as it does not maintain routes or connections that are not required.

[5] This id includes the hash of the MAC address of the first network device and the UUid of the root partition coupled with the process id.

Features such as an option to proactively build a full mesh between nodes in the system and providing actor lookup by name are not available yet, but planned for future releases. The first can increase message passing performance in distributed systems and lessen the dependency on intermediate nodes for routing messages, while the latter improves reachability as names can be used to look up specific actors and can hint at their functionality.

4.3.3 Reachability & Rendezvous on the Internet

In a more general sense, reachability is not always a given in Internet-wide communication as nodes may be hidden behind firewalls or NATs. The Session Traversal Utilities (STUN)[49] allows to detect the public IP address and port in use when located behind a NAT. This requires a STUN server located on the outside that answers STUN requests with IP address and port as seen from its location. Locating a STUN server is possible via a DNS query. This technique can be used to enable direct communication among multiple nodes behind NATs by "punching holes" [50]. Since this is not always possible, Traversal Using Relays around NAT (TURN) [51] defines an extension to STUN to enable communication via a relay server.

The Interactive Connectivity Establishment (ICE) [52, 53] makes use of STUN and TURN to establish connectivity for offer-answer protocols. Nodes that want to connect using ICE are required to communicate via another protocol, e.g., the Session Initiation Protocol (SIP) [54] with allows the offer-answer exchange using the Session Description Protocol (SDP) [55]. Both nodes collect their available address-port combinations, e.g., from their own interface, the NAT and a TURN server. ICE proceeds to systematically try STUN requests for each pair until it finds a working combination to establish connectivity.

HTTP tunneling [56] is another technique to traverse NATs, firewalls or proxy servers. It encapsulate traffic in HTTP requests and responses over port 80 to enable communication that would be blocked due to protocol or port restrictions otherwise, for example through deep packet inspection (DPI). The IPsec tunnel mode [57] can achieve similar tunneling while providing authentication, integrity and confidentiality during transport.

4.3.4 Discussion

The straight forward approach to initiate connections between nodes in actor systems is by knowledge of contact information, commonly host and port. This is often used as an entry point into the actor system. Thereafter, other participating nodes can be met in various, implementation-dependent ways. For example, the proactive creation of a full mesh network introduces nodes to each other node in the system. Alternative approaches such as routing or the reactive establishment of communication channels reduce the number of open channels by only contacting peers when necessary.

Actors are usually propagated by exchanging their references in messages. Depending on the implementation, these references can includes only basic information such as the node and actor id which require additional action by the RE or enough information to locate the actor in the system without additional processing. An example for the latter would be a full paths including a transport protocol, host and port as well as an identifier on the local host. Lookup by name is another common mechanism that usually requires a context such as host information to avoid ambiguous references.

Besides identification of communication partners, messages must be routed through the network to reach the addressed actor. The functionality for this can be split across different layers. The overlay created by CAF nodes currently includes functionality to forward messages based on node ids between nodes that do not have a direct communication channel. This enables actors to reach each other via multiple hops at the cost of complexity. Since the IP layer already provides routing on IP addresses, the RE could rely on its functionality completely instead of introducing intermediate hops. Having a single node with multiple network interface into different separated networks is a case that is handled more easily by the hybrid solution and would require explicit handling when routing is located completely in the underlay.

NAT and firewall traversal have not been considered yet. Although some methods can be used without modification of the framework such as a separately configured HTTP tunnel, providing framework support is a more desirable solution as it could be deployed dynamically without need to for special knowledge by users. A lot of use cases do not require the use of either. To avoid overhead such functionality should be optionally available at compile time.

Actor migration poses some challenges regarding reachability. Propagating a new locations for an actor imposes a data race when a message addressed to it is already in transit. Returning an error bundled with the new location would require the sender to buffer all messages until receipt is acknowledged or returning the full message with the error. Both solutions are undesirable as the first leads unforeseeable buffer requirements for all messages to non-local actors and the second puts additional load on the network. Alternatively, the previous host could forward messages and return an error to update the location information at the sender. For this purpose, the new host has to buffer messages until the migration process is complete. Moreover, lazy propagation of location updates might lead to problems if the previous host failed in the meantime which might lead to isolation of the migrated actor, i.e., its references would be invalidated although still available.

Tasks that use actor migration could be implemented with remote actor creation at the cost of additional work by the developer, e.g., by creating a remote actor, initializing it with the state of the another actor and updating existing references to point to the actor.

4.4 Security

A selected few applications run in completely protected and controlled environments. Tightly coupled clusters and some servers fall into this category. Their networks aren't accessible by the public but may have gateways with restricted access. However, deploying software in such scenarios may also require special network stacks to work over specialized hardware such as InfiniBand. VPN tunnels between distributed and trusted locations provide a lighter coupling. As they require defined endpoints they fit scenarios with low mobility, but allow for distribution in contrast to clusters clusters. While these scenarios do not require security to be build into the framework, many others do. Depending on the application, confidentiality, authentication and integrity are required when exchanging messages. The communication layer of CAF is open to attacks in several ways.

First of all, communication between nodes defaults to unencrypted messages exchanged via TCP / IP. Thus, when communicating via a public network such as the Internet or a wireless network, attackers with access to the network also have access to the exchanged data. Even on seemingly safe desktop computers communication between processes could be monitored by users or a malicious application.

Closely related is the need to authenticated nodes and the origin of messages. Without authentication any node can join a distributed actors system—provided it obtains the necessary information. Once joined, it can capture information, use services offered by actors as well as create new actors or kill existing ones. In the same way, authentication prevents other entities in a network from sending forged messages.

Authenticating nodes is not enough to control access as nodes that are part of an actors system are free to send control messages to other nodes. This enables them to spawn new actors on remote nodes and to kill existing ones, which can easily be abused to shutdown nodes. User clients, for example may not be trusted to have such an amount of control when running in untrusted environments.

The deployment of cryptographic solutions is strongly dependent on the application scenario and link characteristics. Determining the requirements to secure a specific application is the responsibility of the developer. CAF can support this by providing an exchangeable transport layer to adjust the framework accordingly. The access control of CAF messages does, however, reach into the responsibility of the runtime environment. As such, CAF should enable the deployment of interfaces to block harmful messages.

4.5 Scalability

Building a scalable system requires engineering every partial aspect of the system. There has been extensive research to improve the scheduler of CAF to enable scalability up to many cores [9] while retaining a small memory footprint. Distributed scenarios have also been considered to examine how the high-level of abstraction offered by CAF impacts performance compared to low-level APIs such MPI [see 7, sections 2.3] and how CAF compares to other implementations of the actor model [see 7, sections 7.6] in a simple distributed calculation. These benchmarks provide a quick look at the prior performance of the stack and can be considered during the evaluation of this work.

Scalability addresses the adjustment of the system when adding or taking away resources. CAF should scale up to many cores in a local system, up to many nodes in a distributed system and down to embedded devices. There are several dimensions the framework has to handle to scale well.

Actors & Nodes The number of nodes is an dominant factor when considering scalability of the network layer. Each nodes host its own actors and naturally requires communication with other nodes to manage work in the system. In the same way, spawning more actors leads to more messaging and thus may increase demand of the network layer.

Resources Resource availability and the desired usage differ heavily between scaling directions. Scaling up to large system requires using the available nodes to full capacity with regard to processing power and available memory. In addition, applications may be required to extend dynamically depending on workload or demand by temporarily acquiring new nodes. In contrast, scaling down requires applications to get by with limited memory, slow CPUs and limited battery power. The fewer resources are required the cheaper the hardware that can be used for production, which is often of major interest for mass production.

Capacity The framework is also required to scale with the number of parallel connections multiplexed by a node, i.e., the number of its communication partners. While communication does not necessitate an open connection, information about remote actors and runtime environments are tracked locally. Managing these information effectively and without performance impact even for a large number of peers is required. On a constrained node, the available resources may be limiting factor as well.

In addition to application and deployment characteristics, the implementation of the network layer affects its scalability. The transmission process itself introduces memory and operational overhead depending on the guarantees of the transport protocol in use. For each connection it might be necessary to restore the order of received messages, observe delivery to retransmit lost messages or to manage various streams. Deploying a security protocol introduces further overhead to encrypt and decrypt message, check their integrity or to validate signatures which include the management of certificates. These operations do not only require local memory and processing power, but affect message sizes as well.

On top of the transmission process, messages have to be constructed. Since CAF uses its own application layer protocol, its messages needs to be (de)serialized from and to the encoding format used for transmission. Moreover, the protocol headers have to be written and parsed. Choosing an encoding format that fits the task and carefully designing headers for the protocol determines how much processing is need at each endpoint. Both decisions also affect the message sizes during transport. Depending on the environment, different characteristics might be desirable; when scaling up throughput might be more important than message sizes, favoring formats that are quick to parse, while constrained environments can prioritize reducing the size messages.

Tracking routes to actors on non-neighbor nodes adds control plane overhead to message passing. A local information base has to be organized to allow for efficient routing. This requires identifying routes to new nodes, resolving redundant information as well as removing broken ones. Building a full mesh instead of keeping routing information shifts the overhead to the automatic management of connections and synchronization with peers. Information about nodes need to be shared build connections and maintain the network. While this furthers resiliency, the runtime environment may be required to track information and manage connections it never uses.

Finally, state is shared to manage actors and nodes. The resources required to tracking remote entities grows with the number of peers in the system. This includes state for local actor proxies as well as the resources required to monitor nodes to notice failures and propagate them. Deploying name servers adds further overhead to ensure information are up-to-date and provide efficient access to the utility without resulting in a bottleneck for communication in the system.

4.6 Transport Binding

Binding to a specific transport protocol affects messaging guarantees and may limit deployment. Choosing a protocol that has few guarantees might lead to a lot of work to implement functionality granted by other protocols while using a protocol that has many guarantees might impact performance and introduce operational overhead.

Decoupling the network layer from a specific transport protocol enables deployment in many different scenarios and allows optimization for specific use cases. This is not necessarily as simple as exchanging the implementation because a set of behaviors should remain valid to enable transparent deployment.

A suitable interface should abstract over the network layer to allow the exchange of transport protocols as well as adding application layer protocols or module to ensure that the guarantees are kept in place—and potentially allow implementation of stronger or weaker guarantees when desired. Adjusting this way requires the network layer to handle datagram as well as stream-based protocols transparently. The following list presents possible building blocks for our network layer:

UDP Most bare-bones and with little guarantees, the User Datagram Protocol (UDP) [58] is a connectionless and unreliable datagram protocol on the transport layer. As such, it does neither guarantee delivery nor maintain order or detect duplicate packets. This simplicity allows the protocol to work without keeping state at either endpoint. The header only includes port information, the payload length and a checksum for integrity checks. Typical deployments scenarios are environments that have strict timing requirements and cannot wait for retransmits or error correction as well as constrained environments where processing power is limited resource.

TCP The Transmission Control Protocol (TCP) [59] is a connection-oriented protocol with strong reliability guarantees. Packets are delivered in FIFO order between two processes with guaranteed delivery using acknowledgements and retransmits. Additionally, TCP filters duplicate packets. As a stream base protocol, TCP delivers data as a (continuous) stream of bytes rather than in form of datagrams. Due to these guarantees, many protocols are build on top of TCP, provided they can accept the complexity and timing constraints. Examples include HTTP, SMTP, SSH and many more.

SCTP A datagram based protocol like UDP, the Stream Control Transmission Protocol (SCTP) [60] provides stronger guarantees. Datagrams are delivered reliably and can be send via different message streams with guaranteed per-stream delivery order and thus no head-of-line blocking between streams. Furthermore, SCTP provides congestion control and supports multihoming.

Although it was developed with Public Switched Telephone Network (PSTN) messaging in mind, arbitrary data can be delivered. In the absence of native support, SCTP can transported via UDP [61].

QUIC Quick UDP Internet Connections (QUIC) [62] is a young protocol that has been submitted to the IETF, but has not been adopted as of the time of this writing (July 2016). UDP is used as an underlying transport protocol to address compatibility and legacy systems. Key features of the protocol are low latency version negotiation and connection establishments. Similar to SCTP, multiple streams can be multiplexed as part of a single connection which allows the avoidance of head-of-line blocking between streams. The congestion control and loss recovery mechanism aim to be improvements over the mechanism used in TCP.

Security is integrated in the protocol to provide authentication, integrity and encryption. While QUIC included its own cryptography in earlier versions, it currently depends on TLS 1.3 [63]. The protocol also authenticates its headers and encrypts most of them to ensure the protocol can evolve without regard to middleboxes. Although the transport of HTTP/2 is seen as a major use case of QUIC [64], the protocol can be used to transport arbitrary data.

TLS Security for reliable transport layer can be achieved by use of the Transport Layer Security protocol (TLS) [19]. It offers encryption and integrity as well authentication of peers during the handshake. As of now, TLS version 1.2 is standardized while TLS 1.3 is an adopted draft [65] of the Transport Layer Security working group of the IETF.

DTLS Adopting TLS to unreliable datagram protocols requires handling message loss and packet reordering during the handshake. This is achieved by the Datagram Transport Layer Security protocol (DTLS) [66]. Similar to TLS, it provides encryption and integrity. As such, it introduces state and operational overhead such as handshakes.

CoAP The Constrained Application Protocol (CoAP) [67] is an application layer protocol designed for machine-to-machine (M2M) communication in IoT scenarios. It defines a request-response model adapted from HTTP that is optimized for constrained networks. Unlike HTTP it works asynchronously via datagram protocols, such as UDP.

CoAP defines its own messaging layer to deal with the unreliability and asynchronous nature of datagrams. There are two message types to determine the delivery guarantees of a messages. The Non-confirmable (NON) message is simply a fire and forget message while Confirmable (CON) messages use retransmits and acknowledgements.

HTTP The Hypertext Transfer Protocol (HTTP) [68, 69] is a widely used stateless request-response protocol used on a great scale in the Internet. The protocol is usually used over TCP/IP on port 80 with TLS for security.

WebRTC Short for Web Real-Time Communication (WebRTC) [70], the protocol is intended to enable browser-to-browser or browser-to-x communication not only for data, but also for audio and video. The protocol is noteworthy as it transparently establishes end-to-end communication between two endpoints that may be located behind NATs or firewalls. A variety of protocols is used for this purpose [71].

Data is serialized into a defined encoding format before it is transferred over the network. This allows a receiver with knowledge of the encoding to deserialize the data and process it. A general challenge for data formats is the encoding of types. Basic type such as integers, booleans and strings as well as structures structures such as arrays or list are often supported, but knowledge of the object types and structure is required at all endpoints. Some programming languages provide their own object encodings and allow transfer of generic objects, as seen in the Java object serialization. C++ does not include serialization functionality as part of the standard library and requires the developer to handle the process.

The JavaScript Object Notation (JSON) [72] is a popular encoding that has replaced the Extensible Markup Language (XML) in many places. It encodes data in a human readable text format, structured in key-value pairs. Based on JSON, but optimized for size and processing requirements, is the Concise Binary Object Representation (CBOR) [73], a binary encoding format developed for constrained environments.

4.6.1 Discussion

Exchanging the transport layers provides the opportunity to rely on the guarantees provided by the individual protocol. The strengths of each protocol should be examined carefully and should be maintained during usage. For example, utilizing the stream multiplexing of SCTP or QUIC could help to prevent head of line blocking for control and error messages or be used to spread actor communication over multiple streams for similar reasons. WebRTC on the other hand, is an interesting showcase for the transparent establishment of end-to-end connections and should be viewed as an example how to achieve such functionality.

CAF cannot use the serialization of its programming language as it is written in C++, which does not include those capabilities in the STL. Instead, the framework implements its own serialization layer that encodes objects in a binary encoding prepended with type annotations. While serialization of basic and CAF types is included in the framework, users that want to transfer their own classes need implement functions to encode their types and register them in runtime environment. This process limits message exchange to other nodes that run CAF and "know" the type annotations and their types. Otherwise the deserialization process will fail. Brokers give developers access to low-level I/O and enable implementation of other encoding formats or interfaces, for example, to enable message exchange with other languages via ProtoBuf [74].

Considering the focus on scalability, transport mediums with specific use-cases should be considered. High-performance computing (HPC) usually requires clusters with thousands of cores. The prevalent technology to connect those clusters is InfiniBand which offers high throughput at low latency. Although abstractions exist to run TCP/IP over InfiniBand, achieving full performance requires usage of the related API.

Scaling down to constrained environments, IPv6 over Low-power Wireless Personal Area Networks (6LoWPAN) [75] enables Internet connectivity on constrained nodes by simplifying IPv6 functionality and optimizing related protocols. It is often used in conjunction with IEEE 802.15.4 [76], which specifies wireless embedded radio communication.

5 Designing a Network Stack for CAF

Location-transparent communication between actors entrails bridging the network, which is inherently unreliable and provides weak messaging guarantees. The runtime environment can rely on standardized transport protocols in addition to custom application layer protocols to adjust to the characteristics of the network. However, transport protocols offer more than messaging guarantees. Instead of being selected by an application for its guarantees, a transport protocol should be selected to suit the deployment environment. As a result, relying solely on their guarantees is not enough to provide a consistent messaging model.

Establishing guarantees independent of the transport protocol leads to redundancies and processing overhead. Instead, the runtime environment could offer various modules that address a specific guarantee each and deploy a suitable composition depending on the transport protocol. This allows CAF to offer a baseline of functionality and offer a predictable behavior to developers. Different layers of the network stack can host such functionality, starting with an application layer protocol on top of the network stack and going down to modules closely coupled to individual transport protocols.

The routing capabilities of the CAF overlay network require reconsideration. The current design allows forwarding of messages via multiple CAF nodes to reach the addressed actor and thus provides basic routing capabilities. Transport protocols only provide guarantees to single-hop communication between two nodes and extending their guarantees to multiple hops is a complex task. This chapter starts with a discussion of the capabilities of the overlay in section 5.1. Section 5.2 examines what data is passed between the components that make up the network stack in CAF. Next, Section 5.3 proposes API changes to make the new functionality accessible to developers and Section 5.4 discusses how to introduce exchangeable transport protocols into the network stack. Section 5.5 then discusses how the design can be applied to the software design of CAF. Finally, Section 5.6 discusses the achieved benefits, the performance impact and open questions.

With the goal to enable use of arbitrary transport protocols, this section considers TCP and UDP as examples for the discussion.

5.1 The CAF Overlay Network

The main responsibility of the CAF runtime environment is scheduling actors and enabling them to communicate transparently via the network. The two scenarios depicted in Figure 5.1 represent the simple cases the RE has to handle to enable actor communication:

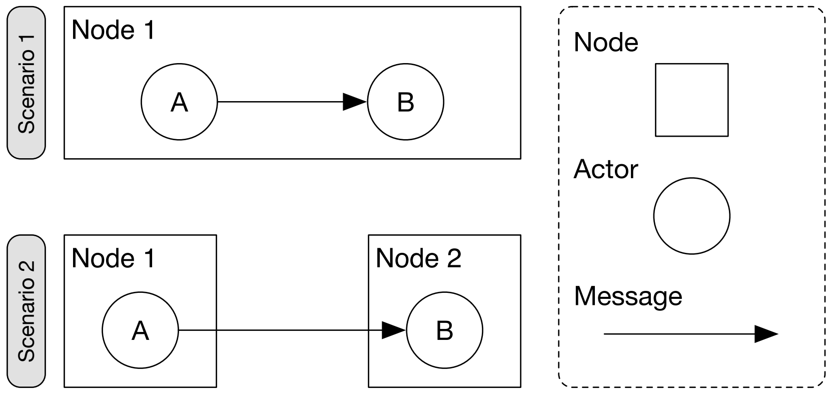

Figure 5.1: Two simple communication scenarios between CAF actors.

1. Actors A and B are located on the same node. Their communication works without involvement of the network layer. Communication failure most likely means that either one of the actors is no longer alive or that the node fails as well.
2. The actors are running on separate nodes with an established communication channel which introduces the network as dependence for message exchange. Their communication is interrupted in case of a network failure and the actors will consider each other failed. Within a limited time frame recovery of the connection might be possible.

The number of nodes participating in an actor system is not limited. Even small distributed actor systems quickly exhibit complex communication topologies. Building a full mesh as a precaution is an easy solution to reduce the management overhead at the cost of keeping state per connection. However, many nodes may never use many of the channels they maintain. Establishing communication channels only once necessary could reduce the required state and keep the management overhead to a minimum. To retain part of the robustness gained from a full mesh, CAF could keep a few additional communication channels to avoid splitting the system if a single node fails. Such an architecture would require an efficient creation of communication channels to avoid delays at runtime.

Figure 5.2 shows a simplified example for a scenario that does not build a full mesh, but remains a single system when a node fails. It depicts four nodes running one actor each.

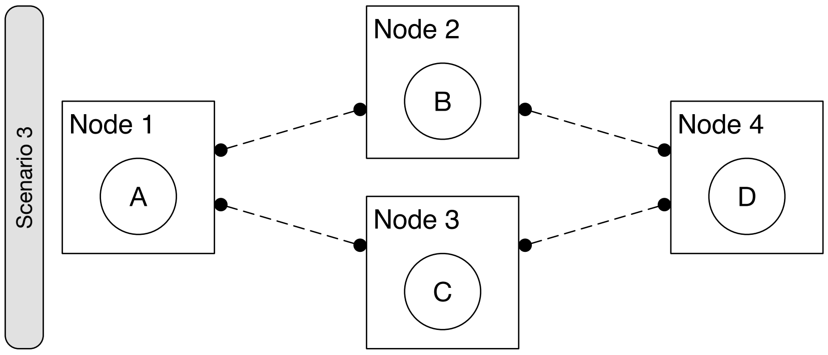

Figure 5.2: A distributed actors system that does not build a full mesh network.

Each node maintains communication channels to two of the remaining three nodes in the system. These channels are part of the overlay, the physical architecture could have all nodes connected via the same switch or distributed through the Internet. Communication between actors on neighboring nodes is straight forward and works in the same way as in the previous scenarios. A new situation is established when an actor learns of the existence of an actor that is not located on a neighboring node, e.g., actor A could receive a message from actor B that contains a reference to actor D. If actor A then addresses actor D in a message, the RE is responsible to deliver the message transparently. Following, two approaches to routing messages are discussed.

In the following discussion, *hop* refers to an intermediate CAF node that may forward messages not addressed to local actors.

5.1.1 Routing in the Overlay

Message delivery to actors on non-neighboring nodes could be handled through routing in the CAF overlay. Nodes forward received messages which do not address local actors according to routing entries held in a local routing table. This table is build reactively when a node receives a message that contains a reference to an actor hosted on an unknown node. Routing entries consist of the new node id and the node id of the neighbor that delivered the message. Messages addressed to actors on the new node are routed via said the neighbor subsequently. This mechanism assumes that the neighbor which delivered the message in the first place knows a valid route.

For example, actor A wants to send a message to actor D. The RE would send the message to the node that propagated the existence of node 4 earlier, e.g., node 2. After the first hop, the node 2 would find that the message does not address a local actor and forward the message according to the node id in the message header. In this case, to node 4 which hosts actor D.

As part of the design, messages between a pair of actors can take multiple routes. A failure of node 2 (see Figure 5.2) does not necessitate that nodes 1 and 4 cannot communicate any longer. If nodes 1 and 4 are aware that the other node can be reached via node 3, they could try to use that route before assuming failure. Maintaining a routing table and maintaining connectivity in the presence of failures requires significant maintenance by the RE.

Detecting node failure in such scenarios requires more than the observations of a single node as the communication between nodes 1 and 2 might be interrupted while nodes 2 and 4 are still able to exchange messages. In such a scenario, node 2 could propagate the failure of node 1, requiring the distributed actor system to form consensus and update routes to enable communication between actors on the affected nodes. For this purpose, not only used routes but available routes would have to be tracked. Managing such a network introduces a high amount of complexity into the overlay.

In many scenarios, communication failure cannot be resolved by routing messages via another node in the overlay as nodes may be connected via the same infrastructure, e.g., in closely coupled environments. In other cases, such as the failure of single router in the Internet, retransmission of the message can be enough to solve the failure as the routing is adjusted by the IP layer without attention of the CAF runtime environment. Redundant routing paths can also be handled on the transport layer by the Stream Control Transmission Protocol (SCTP) [60] or Multipath TCP (MPTCP) [77] which extends regular TCP to utilize multiple paths simultaneously.

5.1.2 Routing in the Underlay

Routing is an essential part of communication in the Internet and handled by layers below the network stack in CAF. Instead of implementing routing capabilities in the overlay, CAF could rely on the IP layer for delivery between nodes. This would remove message forwarding from the responsibilities of the RE. Looking at Figure 5.2, communication between actor A and D would be enabled by establishing a new communication channel using host and port information of the destination node. When node 1 receives a message containing the actor id of actor D, it would ask the sender for contact information such as host, port and a transport protocol to exchange messages directly, in this case with node 4 which hosts actor D.

Nodes could exchange contact information of peers proactively to increase the robustness of the distributed actor system. In case of node failure, this could prevent nodes from being isolated. Instead the RE could use the cached information to contact other nodes and attempt to recover from the failure. While unreachability of a node would still result in unreachability of all of its hosted actors, the rest of the system could continue to function.

Technologies such as NATs and firewalls may hinder the creation of a communication channel. Configuring the transport layer to use standards such as ICE directly or in the form of a transport protocol, e.g., WebRTC, would provide a reasonable solution to enable connection establishment for most scenarios. Propagating such failures could be clarified by introducing a appropriate failure type and distinguish them from failures of nodes or established communication channels.

Management of such an architecture is easier compared to the overlay routing approach described previously. It does not require routing across multiple hops (i.e., intermediate CAF nodes). Instead transport works end-to-end between CAF instances and the guarantees it offers do not have to be extended but can be relied on instead. Special cases such as the reestablishment of broken TCP connections still require management, although at a reduced complexity.

5.1.3 Discussion

Routing messages via multiple CAF instances is currently implemented by CAF. The RE does not extend the guarantees of the transport layer to multiple hops for this process and instead introduces the risk that messages arrive out of order or get lost.

When introducing new transport protocols that do not provide guarantees on their own such as UDP, reliable ordering and delivery have to be established by the RE. While this could easily be implemented between any two nodes in the system, running these modules independent of the transport layer introduces considerable overhead in cases where the transport protocol already implements similar functionality.

Since routing is already implemented on the underlay, the benefits of dedicated routing implementation in CAF does not provide much benefits. Considering the complexity that changes or failures in the routing topology introduce, making a case for overlay routing becomes even harder. In contrast, relying on the routing capabilities does not only simplify the implementation, but enables CAF to fully benefit from the guarantees of standardized transport implementations such as the reliability guarantees of TCP or the connectivity establishment of WebRTC. NAT traversal with TURN [51] requires a relay server that is reachable from all nodes hidden behind NATs that want to communicate. To provide a seamless deployment, CAF instances would have to work as TURN assistants in such scenarios.

As a result, the redesign of the network stack focuses on the establishment of node-to-node communication while relying on routing in the underlay. While CAF will no longer require routing tables for message forwarding, new components are required to dynamically exchange and cache addressing information with neighbors to enable communication with remote nodes when required.

5.2 Data Flow in the CAF Network Stack

The network extension in CAF consists of several components that manage network communication including the routing capabilities currently present in CAF. The middleman provides an API to developers and manages the brokers. Figure 5.3 shows the data flow between the components BASP, broker, multiplexer and the socket API:

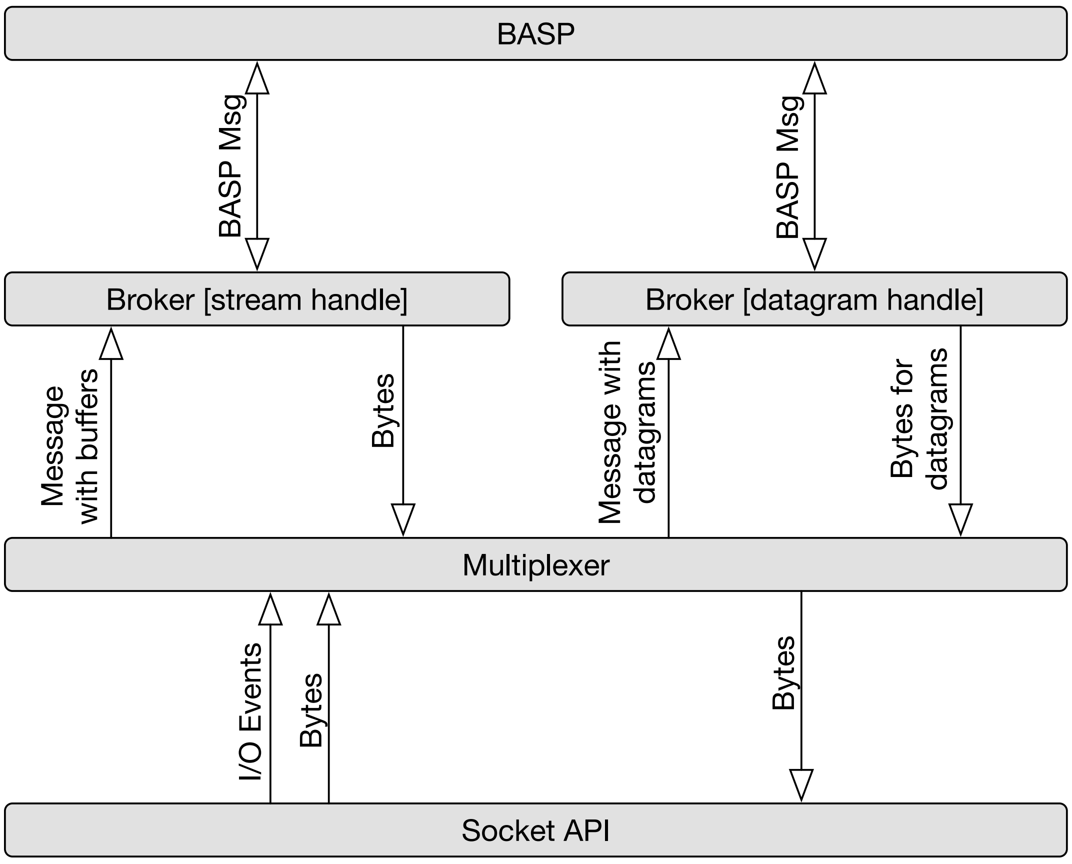

Figure 5.3: Data exchanged between CAF components in the network layer

BASP This application layer protocol is used to manage distributed CAF nodes. As such, it encapsulates messages exchanged between actors, errors that are propagated in the system and enables functionality such as remote actor creation. BASP messages are processed by brokers.

Before the redesign, the BASP included the IDs of both the sender and receiver nodes. These were used for routing based on forwarding messages not addressed at the local node, thus allowing multi-hop transmission.

Brokers These components provide an abstraction for the upper layers by translating node IDs to handles which identify via which protocol messages are sent. Brokers also (dis)assemble BASP messages to and from bytes. Assembled BASP messages are passed upwards, while bytes are exchanged with the multiplexer. Brokers call functions to directly write bytes into buffers offered by the multiplexer and received actor messages containing byte buffers in return.

For stream-based communication, a broker can pass a continues stream of bytes to the multiplexer. When reading, brokers have to parse messages boundaries from the incoming byte stream. An example is the previously existing implementation based on TCP which uses these brokers.

Working with datagrams requires additional management of a broker as the data cannot be sent as a stream but in packets. This requires slicing outgoing messages according to size limitations before passing them on to the multiplexer as bytes fitting into a datagram. As a consequence, incoming datagrams potentially have to be sorted and combined to rebuild BASP messages.

Multiplexer Mapping handles to the associated protocol and socket, the multiplexer sends the bytes it receives from brokers to the network. Additionally, I/O events received from the socket API allow the multiplexer to observe multiple sockets in parallel and react on incoming data. The bytes it reads this way are passed upwards to the brokers.

The multiplexer instantiates components called “doorman” and “scribe” per socket to handle these tasks. As their functionality is specific to TCP, adding new protocols requires a protocol specific components that can be used by the multiplexer subsequently.

Sockets This API is an interface offered by the OS to enable communication between processes, locally and in the a network. It is used by the multiplexer for reading and writing bytes.

5.3 Application Programming Interface

The use of a specific transport protocol for one communication channel should not dictate what protocol is used for all communication on the node. Instead, choosing a protocol should be possible per individual peer and depending on the deployment. This requires developers to be able to specify the protocol when publishing actors or contacting remote nodes. The related API offered by the middleman expects the host as a string and a port as an integer. Exchanging these parameters with a URI enables configuration and allows for backwards compatibility by mapping the existing functions to the new ones using the previous default transport protocol, i.e., TCP. This introduces a new source of errors when an unsupported protocol is specified as a scheme. Listing 5.1 shows a simplified API of the affected functions, which are available via the middleman. For the functions `publish`, `remote_actor` and `spawn_client` these changes are straightforward. The remaining two functions (`unpublish` and `spawn_server`) did not have an argument for the host previously, but require one now as an URI cannot contain a port without host information. Additionally, a scheme must be included in the URI. This is of significance as different actors might be published on the same port with different protocols.

```
/// Tries to publish actor at port and returns either an
/// error or the bound port. Passing no address sets the
/// IN_ADDRANY option. The last argument determins whether
/// the flag SO_REUSEADDR is set.
uint16_t publish(actor, port, address, reuse);
uint16_t publish(actor, uri, reuse);

/// Unpublishes the given actor by closing port or all assigned
/// ports if 0 is passed.
void unpublish(actor, port);
void unpublish(actor, uri);

/// Acquire a proxy of the actor at host on a  given port.
actor remote_actor(host, port);
actor remote_actor(uri);

/// Returns a new functor-based broker as a client for the
/// server at a given host:port or an error.
broker spawn_client(functor, host, port, arguments);
broker spawn_client(functor, uri, arguments);

/// Spawns a new broker as server running on a given port.
```

```
broker spawn_server(functor, port, arguments);
broker spawn_server(functor, uri, arguments);
```

Listing 5.1: Using URIs instead of host and port arguments.

A wild-card scheme could be used to allow late binding to a transport protocol. For example, when contacting a published actor on a remote node, the RE could try the available protocols in a configurable order until either the contact was successful or all available transport protocols were tried out. A challenge with such an approach is choosing timeouts that fits the protocol, e.g., when using connectionless protocol such as UDP, and the deployment. The schemes `any://` or simply a star (`*://`) could be used for this purpose in URIs.

Reliable message delivery is not only of interest for internal management messages such as error propagation and the related link and monitor messages, but might be desirable in various use-cases. Since the guarantee introduces overhead when using a transport protocol that does not provide this guarantee per default, sending a reliable message should be explicitly enabled through the messaging API. Duplicating each function and adding a postfix such as `\_reliably` does not only lead to a large number of functions, but additionally is not extensible for future adjustments. An alternative approach would be a template argument. Since CAF already allows passing priorities as template arguments, this is a viable solution that extends a familiar configuration method.

An example for sending a reliable message is shown in listing 5.2. First, a URI that addresses a non local actor is created and passed to the function `remote_actor` to acquire a local proxy of the addressed actor. A local actor called `sender` is created to send a message to the remote actor, called `receiver`. Line 7 shows how to send a message with the guarantees given by the transport protocol, in this case UDP as indicated in the URI. In contrast, the message in line 8 is send reliably by passing the template argument `guarantee::reliable` to the send function. Adjusting guarantees of other function that send messages (`send_as`, `delayed_send`, `anonymous_send` and `request`.) works in the same way.

Reliability is one of the guarantees CAF aims to provide across transport protocols (discussed in the following Section 5.4). Hence, every protocol that can match a wild-card URI either offers reliable delivery or CAF deploys the functionality on top.

Unlike reliable delivery, ordering cannot be decided per message. When receiving a message marked as ordered, a context would be necessary to determine other message that might not have arrived yet, but belong into the order. Although additional fields in message headers could give these information, this adds overhead to all messages. As discussed in section 4.2.2, CAF aims to provide FIFO ordering for all messages. This ordering will be applied to all messaging

```
auto u = uri::make("udp://mobi42:1337");
// acquire a local actor proxy of the remote actor addressed by u
auto receiver = remote_actor(*u);
scoped_actor sender;
// Send a message with the guarantees of the transport protocol
sender->send(receiver, my_atom::value, "Hello");
// Add reliable delivery, independent of the transport protocols
sender->send<guarantee::reliable>(receiver, my_atom::value,
                                     "World");
```

Listing 5.2: Introducing a template argument to adjust messaging guarantees.

and does not require a specific flag. In listing 5.2, this results in `"Hello"` being delivered before `"World"` unless the first message is lost.

5.4 Designing the CAF Network Stack

Relying on the routing capabilities of the underlay (see section 5.1), CAF no longer needs to handle message delivery via multiple hops. Instead, the framework relies on the guarantees provided by transport protocols. The set of basic guarantees we want to provide consistently are:

1. Reliable delivery (optional)
2. Ordering (FIFO)
3. Detecting failure of peers

As the network stack should be able to handle different transport protocols, TCP and UDP will be considered as examples during the design. Looking at TCP, the first two guarantees are already part of the protocol and the third one can be achieved by observing its connection state. Reliable delivery can be strengthened even further by adding a component that tries to extend the guarantee over TCP reconnects. In contrast, using UDP requires the RE to add the missing functionality. As noted in section 5.2, brokers require varying implementations depending on the transport protocol to handle the protocol specific packet requirements, in the case of TCP vs. UDP either using streams or handling buffers to satisfy transmission via datagrams.

While not all functionality of transport protocols is covered by these two choices, they are on the opposing ends when looking at the included functionality. While UDP is a bare-bones protocol that provides connectionless transmission of datagrams with few guarantees, TCP streams data with strong reliability and ordering guarantees, additionally handling congestion control and data slicing. Thus, this protocol selection provides the opportunity to examine how CAF can strengthen the guarantees of protocol as well as how to adapt to existing ones.

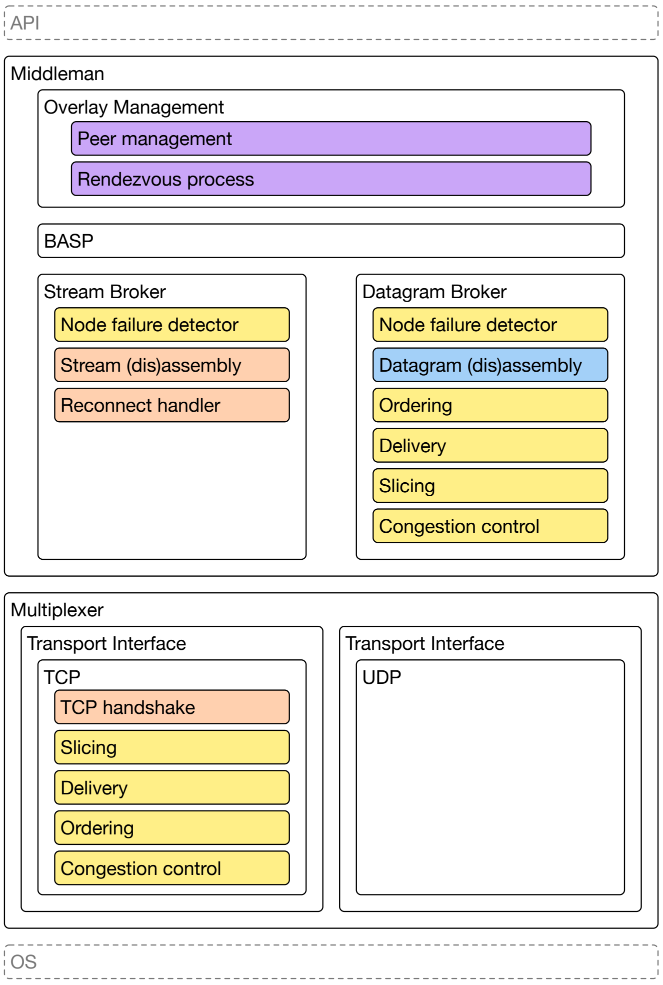

Figure 5.4: Composition of the CAF network stack deploying TCP and UDP.

The design of the network stack is shown in Figure 5.4. The *middleman* distributes is functionality across three components: (1) the *overlay management* exchanges contact information in form of protocol, host and port for remote nodes with neighbors to allow messaging remote actors without routing in the overlay, (2) the application layer protocol *BASP* used to manage the distributed CAF nodes and (3) a *broker* for each usable transport protocol. Brokers translate between BASP messages and bytes. In addition, they instantiate additional components to customize the guarantees of their respective protocols. The box colors determine their affinity to specific protocols. Yellow boxes represent general functionality that should be provided independent of the transport protocol. Orange boxes are specific to TCP and blue boxes are specific to UDP while purple boxes concern the overlay.

In the example case, the stream broker is responsible for TCP. It adds high level functionality to handle failure detection and TCP reconnects in addition to a component to read and write BASP message from and to a byte stream. The second broker handles UDP datagrams. Similar to the stream broker it deploys a failure detector and a component to translate between datagrams and BASP messages. Additionally, it requires a component to slice BASP messages into packets and to order incoming messages, to be able to rebuild BASP messages that were sent in multiple datagrams as well as to satisfy the FIFO ordering guarantee. The delivery component is enables the requirement for reliable delivery, but is only used optionally depending on the user input. A reconnect handler is not deployed as UDP messaging does not have connections.

The *multiplexer* is a low-level component that interfaces with the OS and uses the socket API to receive and sent data. In Figure 5.4 it utilizes the transport protocols UDP and TCP via a common *transport interface*. This interface unifies protocol specifics to avoid adjusting the multiplexer to each individual protocol. The TCP protocol provides general required functionality (slicing, delivery, ordering and congestion control) as well as a TCP specific handshake. In contrast, UDP includes no high-level functionality itself, pushing much more responsibility into its broker.

5.4.1 Design Considerations

Overlay management This new component ensures node-to-node connectivity by exchanging contact information such as host, port and protocol information with peers. Furthermore, a handshake between CAF nodes could be handled by this component. Locating this functionality above BASP in the stack allows the introduction of new BASP messages for these management tasks. Operating solely on top of BASP also provides a separation from transport protocol dependent functionality.

Brokers Extending protocol guarantees in the brokers is a straight forward solution as they already require protocol dependent functionality to translate between the incoming data and BASP. Bundling protocol specific functionality on this layer separates components upwards in the stack from transport dependent logic. An alternative approach could be to use the same guarantee-adjusting components for all protocols. While this would simplify the design, a large overhead would be introduced to all protocols that already implement some of the guarantees. While some components, e.g., ordering or delivery, can be reused for multiple protocols, even components that provide general functionality such as a failure detector can benefit from protocol specific implementations. As an example, a TCP failure detector could monitor the connection state to detect failures and avoid additional heartbeat messages. Protocol dependent functionality is not only required, but can enables a more efficient implementations. Brokers are a fitting wrapper for this task.

Multiplexer As shown in Figure 5.3, the multiplexer exchanges data with brokers deployed by the middleman. Even though the multiplexer uses protocols via a unified interface, the data passesed to the brokers is protocol dependent and requires different processing.

Drawing functionality from brokers into the multiplexer and bundling guarantee-adjusting components with the transport protocol they augment would allow gathering of all transport protocol related functionality in one layer. For example, a UDP-based transport bundle could include components for ordering and reliability guarantees through “pre-sending” and “after-receiving” callbacks in the multiplexer. However, this would require the multiplexer to interpret the received bytes in order to perform these operations and add complexity to the multiplexer. Separating functionality of the transport protocols from framework-specific adjustments separates the concerns and eases the extension of the individual components, e.g., to add new transport protocols or exchange the ordering or reliability guarantees.

Transport Interface The unified transport interface allows the multiplexer to handle various protocols by wrapping them each in a thin layer. Developers that want to integrate a new protocols only have to satisfy such an interface and provide a list of functionality offered by the protocol to allow the broker to handle the missing functionality. The components listed in Figure 5.4 might not be exhaustive, but allow adjusting guarantees to the set CAF wants to provide for actor communication.

5.5 Software Design

Introducing the abstract design (see Figure 5.4) to CAF requires a few independent tasks. The first one introduces URIs and extends the API of the multiplexer to accept URIs as arguments. The second task decouples CAF from TCP and enables the use of other transport protocols. Lastly, the runtime environment has to be stripped of its overlay routing capabilities and instead be enabled to communicate with remote hosts directly.

Figure 5.5 shows the related API in the middleman. The top five functions existed previously and received a new overloaded function that accepts a URI instead of host & port arguments.

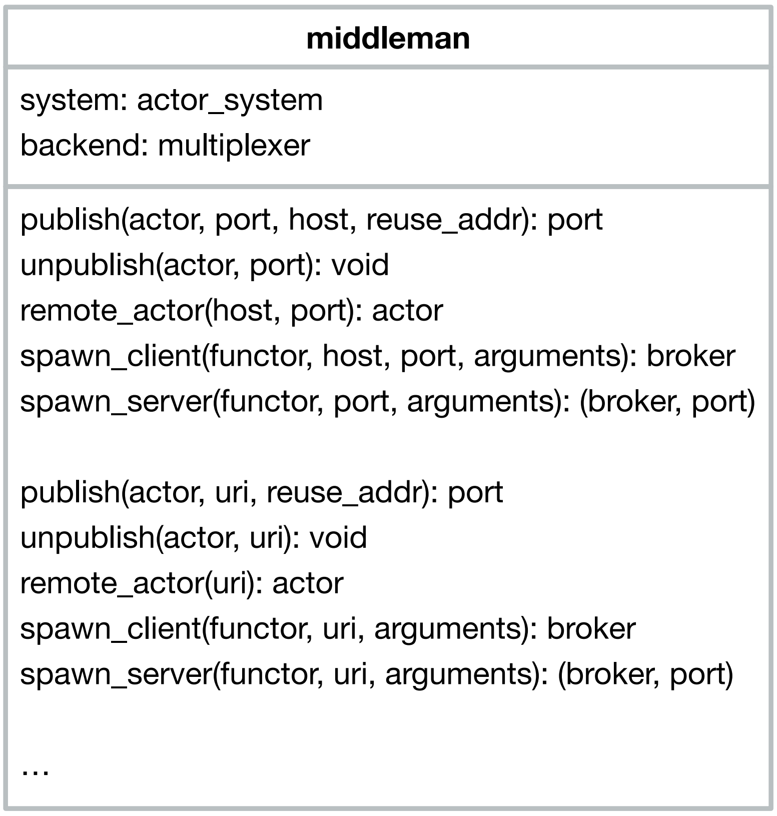
middleman
system: actor_system
backend: multiplexer
publish(actor, port, host, reuse_addr): port
unpublish(actor, port): void
remote_actor(host, port): actor
spawn_client(functor, host, port, arguments): broker
spawn_server(functor, port, arguments): (broker, port)
publish(actor, uri, reuse_addr): port
unpublish(actor, uri): void
remote_actor(uri): actor
spawn_client(functor, uri, arguments): broker
spawn_server(functor, uri, arguments): (broker, port)
…

Figure 5.5: Introducing URIs to the middleman.

Existing functions create a URI object internally and call their overload to ease maintenance in the future. Without introducing new transport protocols to CAF, the only accepted scheme is `tcp` which maps to the existing TCP-based backend of CAF and is used as the default scheme. Once multiple protocols are available, each function accepting a URI as input will dispatch further functions calls depending on the scheme.

Introducing a new transport protocol is a more complex task. Protocols and their interfaces differ greatly, requiring a varying degree of integration into the runtime environment. Easy to integrate are protocols that extend functionality or can be encapsulated in an available protocol. HTTP tunneling between CAF nodes is an example for such a protocol as it only requires altering the payload, but uses the same sockets and calls as TCP. contrast, protocols that differ greatly in their behavior and interface require integration into CAF, as will be discussed with regard to UDP. Before discussion how protocols can be integrated into CAF, the following paragraphs provide an overview over the implementation of TCP.

The management of TCP is divided into two high-level classes (`doorman` and `scribe`) and two low-level classes (`acceptor` and `stream`). The high-level classes are located between the broker and the low-level classes where they offer an interface for sending data over connections and provide incoming data wrapped in actor messages. In contrast, the low-level components handle read and write events and interface with the socket API to send and receive. A `doorman` awaits new TCP connections and interfaces with the `acceptor` component which waits for and accepts incoming TCP connection attempts. The class `scribe` manages an established TCP stream using the `stream` component. Figure 5.6 shows how the classes scribe and stream are related and embedded between the broker and multiplexer. The Figure display inheritance as vertical relationships and interactions as horizontal relationships. Abstract interfaces are written in *italic* font.

The abstract class `broker_servant` defines a high level interfaces and ensures that derived classes can manage their related low-level components. The derived class `scribe` implements the interface used by brokers to sent messages. Each broker implementation is derived from the abstract broker and keeps track of the `broker_servant`s it manages. Derived from the class `scribe` is the local class `impl` that adds the interaction with the low-level class `stream`. Derived from an `event_handler`, a `stream` reads and writes data to and from the network. It is executed by the `multiplexer` which implements an event loop to manage its components.

Data flows from left to right when it is send and in the reverse direction when it is received. For sending messages, a broker looks up the scribe responsible for the connection and passes its data. In turn, the scribe passes the data to its stream which writes the data to the related socket once socket buffer has space available. On reception, the stream is notified when data has arrived on the socket. It accepts the data and passes it to the responsible scribe where the data is wrapped into a local actor message and sent to the broker that manages the scribe.

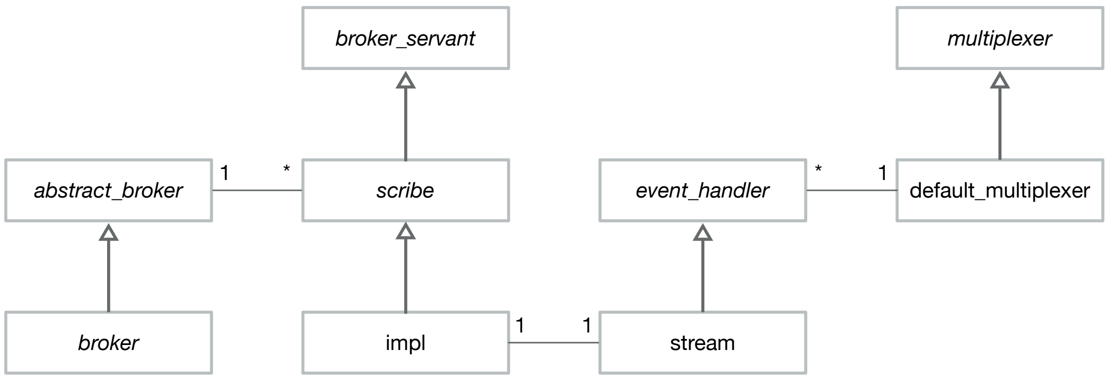

Figure 5.6: The relation between the high level *scribe* and the low-level *stream*.

Implementation of a protocol that uses an existing protocol for transport can be done via protocol policies. A policy configures the behavior of a class and is implemented against an implicit interface. Depending on the design a class that is configured by a policy either holds the policy as a member or derives from it. Figure 5.7 depicts the relation between a class and its policy, using the classes `scribe` and `doorman` as an example. To configure the TCP implementation, a policy for both components would have to be implemented. Each policy defines a set of functions that the configured component calls during processing.

The policies depicted in Figure 5.7 each require a function `copy` that initializes a new policy object with a blank state or returns itself it no state is required for processing. Further, the `handshake_policy` requires a function that returns a new `communication_policy` to pass to the scribe that handles a the connection accepted by the doorman. This allows the doorman to pass on details negotiated during the handshake. The remaining functions allow pre- and post-processing of messaging data.

In this case, each component can be configured with up to one policy—no policy denotes use of the raw protocol. In turn, a policy object can used by any number of components as long as it does no hold state, and must be copied otherwise. In the presence of a policy, a component calls the related policy functions as part of its regular procedure to allow additional processing steps before sending and after receiving data.

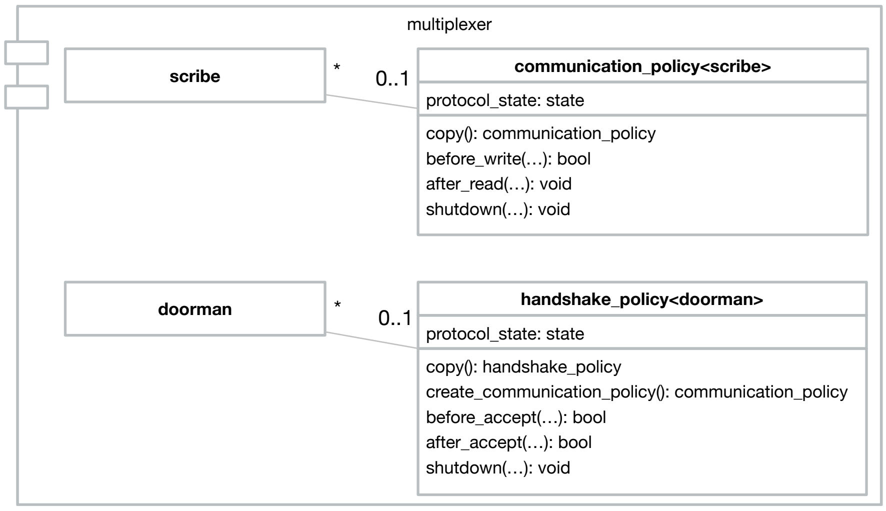

Figure 5.7: Protocol policies extend the functionality of a “raw” transport protocol.

Protocols that do not map to an available “raw” transport protocol require integration into the framework similarly to the TCP integration discussed above. This requires implementation of a component to start communication with a remote endpoint and a component that provides a local endpoint to allow remote nodes to initiate communication. The multiplexer uses the classes already mentioned for TCP: the `doorman` and the `scribe` as a high-level abstraction towards the broker and the `acceptor` and `stream` to handle the low-level events and perform socket operations.

UDP is an example of a protocol that requires a deeper integration as it cannot be implemented as a policy. It uses a different socket type, packs data in datagrams and is connectionless. However, its communication procedure can be separated into client-server roles, similar to the TCP implementation. A local socket is open and awaits contact from a remote socket, and then creates a dedicated socket for each remote endpoint. In CAF this maps the classes `dgram_doorman` and `dgram_acceptor` which wait for contact from new remote endpoints and instantiate a `dgram_scribe` and `dgram_stream` to handle the message exchange with the remote endpoint subsequently. Additional management is required to manage addresses of remote endpoints as well as sending and receiving datagrams as there are no connections these components can rely on. For optimizations, the number of open sockets could be reduced as a single socket can be used to send messages to multiple endpoints.

CAF brokers are a generic component that can be implemented by users of the framework. They provide an actor-based interface for network I/O, e.g., the translation between actor messages and HTTP requests and responses. A broker itself is an actor that reacts to a defined set of messages which are sent by broker servants (e.g., active `scribe`s). The BASP broker—the class `basp_broker` inherits from the class `broker` in Figure 5.6—is the broker implementation responsible for the message exchange between CAF nodes including error propagation, remote spawn and general messages. In addition to messages from its servants, the BASP broker accepts messages to publish actors or initiate contact with remote actors. These messages are sent by the middleman as part of the related API calls.

The design in Figure 5.4 depicts two separate brokers, responsible for TCP and UDP. While this matches the set of messages passed upwards from the servants, the BASP broker is a single entity that takes on both roles—and might get additional functionality for future protocols. This design allows easy sharing of guarantee-strengthening capabilities, once ordering is implemented in the BASP broker for datagrams, other protocols can make use of the functionality as well.

The overlay management is hooked into the BASP broker which maintains a routing table that maps node IDs to handles of the responsible broker servants. As part of the redesign, the routing table will be replaced with an address cache to manage information about peers consisting of node IDs, addresses and local endpoints, should they exist. To exchange addresses with neighboring nodes, the BASP protocol can be updated to include message types that fit task. Addressing remote nodes reactively requires a small cache to store messages until the application handshake is completed. In parallel to message exchanges, the RE can start to share its address data with a new peer and update its local the address cache.

5.6 Discussion

The move away from overlay routing to a dynamic creation of communication channels provides two major benefits. First, it simplifies message delivery by stripping away routing decisions and handling of associated failures. Second, by relying on the routing capabilities of the underlay, guarantees provided by transport protocols can be adapted and—in contrast to routing in the overlay—do not need to be extended to multiple hops.

A new component is responsible to enable communication with peers. This management component can perform an application handshake that exchanges contact information with neighbors to allow address other peers when necessary. Contact information include a transport protocol in addition to host and port information. A single node may be reachable via different protocols and have multiple actors published on various ports. The address cache should address this and allow configuration of a preferred protocol. Upon receipt of a message that includes an actor ID from an unknown node, the runtime environment tries to contact the host node. In case it does not have address information cached, it requests information from the sender. Encoding location information into actor IDs can be considered as CAF does not allow actor mobility. Such a change should considered carefully and examined with regard to efficiency as actor handles are passed around and accessed frequently.

Brokers are decoupled from TCP and enabled to handle arbitrary transport protocols. For this purpose, a broker can instantiate modules that strengthen the protocol guarantees. The modules available to the runtime environment enable usage of TCP and UDP, but can be extended later on, either for optimization to specific environments, or when adding new transport protocols with different requirements. Adding an abstract interface for transport protocols allows the multiplexer to handle each protocol that is wrapped in the interface. This allows bundling information with each protocol, such as a scheme to select it for communication and its features to enable brokers to handle only the necessary guarantees.

Detecting failure of a single neighbor is often addressed and can be implemented with varying complexity. A simple implementation for CAF would either monitor transport layer connections or test liveliness more actively, e.g., by sending regular heartbeats. For larger distributed systems determining node failures further requires consensus as nodes may observe different behavior. This requires further thoughts and a separate implementation.

The default protocol for communication will continue to be TCP, which provides strong guarantees on the transport layer and requires CAF to add little. Since TCP connections can break at times, the delivery guarantees provided by the protocol could be further strengthened by implementing a mechanism that allows the RE to re-establish such broken connections. However, this is an expensive process as it requires buffering messages in the meantime.

The software design is not as straightforward as the general design concept. A unified transport interface towards the multiplexer is not provided for arbitrary transport protocols. Implementing UDP requires deep integration into the framework. The protocol uses its own socket types and requires the BASP broker to handle datagrams. In contrast, implementation of an HTTP tunnel requires an additional step to wrap messages into HTTP. This step can be performed by broker servants while passing data between brokers and event handlers. A protocol policies implements such functionality and configures a servant for the task.

6 Implementation

The implementation of the redesigned network stack consists of the following tasks (1) introducing URIs to the API, (2) adding support for UDP, (3) introducing transport protocol policies and (4) replacing the routing table with an address cache.

This work contributes and addresses task (1) with an implementation of a new URI class and its integration into the API of the middleman. Task (2) is addressed as well. Two new low-level event handlers are implemented in the multiplexer to send datagrams, receive datagrams and handle new endpoints. The event handlers are managed by new broker servants that provide and interface to brokers. Lastly, the BASP broker was extended work with these new component to enable UDP communication between CAF actors.

The remaining two tasks address an extensible transport layer and rework addressing in the CAF overlay network. While the implementations of UDP already opens the the middleman and brokers for the integration of new transport protocols, the implementation of protocol policies is the main contribution for task (3). Removing the routing capabilities from the CAF in favor or reactive addressing (4) reworks the capabilities of the overlay network. These changes should mostly affect the BASP broker and the protocol. Although not yet implemented, both tasks will be addressed in the future.

The code is hosted on GitHub [1] and open source under the BSD 3-clause license, offering the MIT license 1.0 as an alternative. Building CAF requires a C++11 compliant compiler and should work on Linux, macOS, FreeBSD and Windows.

6.1 URIs

CAF utilizes URIs to identify communication endpoints between processes. This requires the *scheme*, *host* and *port* fields of an URI. The implementation parses other fields as well, conforming to RFC 3986 [78]. While URIs are expressed as strings, the class `uri` provides a wrapper that offers access to individual fields. The factory function `uri::make` parses a string

[1] https://github.com/actor-framework

```
auto u = uri::make("udp://mobi42:1337");
if (u)
  auto a = remote_actor(*u);
```

Listing 6.1: URI creation in CAF.

of returns an object of class `optional<uri>` that either contains a valid `uri` or nothing. This can be tested using a boolean expression as shown in Figure 6.1.

The listing first calls the factory function to create URI objects from strings. The keyword `auto` provides type deduction in C++, in this case for type `optional<uri>`. Subsequently, the if-statement tests the optional for its contents, before a call to `remote_actor` access the `uri` in the optional to acquire an actor proxy. Optionals allow access to their wrapped object using the `*` operator while functions of the object can be accessed via `->`, similar to pointer semantics.

The parser uses a single-pass algorithm to process the input string. Each field begins with a special character specified in the standard which makes parsing a straightforward task. Instead of copying parts of the URI string, the class saves a pair of iterators as bounds for each field. The data is not stored in the `uri` object itself, but inside a private object of type `uri_private` that can be shared by any number of objects. This separates the data of the object from its access handle to achieve O(1) copy semantics. Listing 6.2 shows the immutable interface of the class `uri`. It consists of the factory function (`make`) and accessors for individual URI fields. As long as the interface does not offer mutable access to the private members, each private URI can be shared by any number of public objects.

Pairs of string iterators saves memory compared to copying each string component while still allowing string comparisons and conversions. The API changes to the middleman are straightforward as discussed in 5.3.

Going forward, C++17 will allow us to replace the iterator pairs with `string_view` [79]. A standardized solution is preferable as it works well with the rest of the standard library and provides a familiar interface to C++ developers.

6.2 Middleman Adaption

The middleman translates between actor and network messages and provides an interface for network related operations such as publishing actors or acquiring proxies of remote actors. Extending the middleman API to accept URIs as arguments was already discussed in Section 5.5 and implemented accordingly. The middleman deploys an actor of type `middleman_actor`

```
using str_bounds = pair<string::iterator,string::iterator>;

class uri {
public:
  const string& str() const;
  const str_bounds& host() const;
  const str_bounds& port() const;
  const str_bounds& path() const;
  const str_bounds& query() const;
  const str_bounds& scheme() const;
  const str_bounds& fragment() const;
  const str_bounds& authority() const;
  const str_bounds& user_information() const;
  static optional<uri> make(const string& uri_str);
private:
  uri(uri_private* d);
  intrusive_ptr<uri_private> d_;
};
```

Listing 6.2: The public interface of the URI class.

for processing calls and communication with the BASP broker—which only offers a messaging interface. Middleman functions that require interaction with the BASP broker send request to the middleman actor and block until the actor processed the task and sends an answer. Alternatively, messaging the middleman actor directly can avoid the blocking call through the interface of the middleman.

With the addition of exchangeable transport protocols, the middleman has to parse the URI scheme to handle the functionality of `remote_actor` or `publish`. Scheme dispatching was implemented as a simple string comparison to enable UDP functionality. After dispatching, the middleman creates a corresponding broker servant and sends its handle to the BASP broker, which assigns itself as a managing broker. For publish, the middleman actor can return the port of the doorman directly after forwarding the doorman handle to the broker. However, acquiring a handle to a remote actor requires the broker to contact the remote host before returning the handle, unless the remote handle is already cached.

6.3 Enable UDP Communication

Establishing communication via UDP is handled similarly to TCP (see Section 5.5). After publishing an actor, a local component waits for communication request from other CAF nodes. For each request, it creates a new component to handle the data exchange with the new

	Broker Servants		Event Handlers	
Task	TCP	UDP	TCP	UDP
Handle new endpoints	`doorman`	`dgram_doorman`	`acceptor`	`dgram_acceptor`
Exchange messages	`scribe`	`dgram_scribe`	`stream`	`dgram_stream`

Table 6.1: The components handling TCP and UDP communication in CAF.

endpoint. Acquiring a handle to a remote actor works by creating a local component to initiate contact, create the local actor proxy and handle the subsequent data exchange.

Each communication between CAF nodes begins with an application layer handshake that exchanges node identifiers. The UDP implementation opens with the client handshake as a first message and expects a server handshake message in return. The runtime can decide to terminate the communication endpoints once the handshake is complete and it determines that it already has a local endpoint to communicate with the same CAF node. This procedure requires minimal changes to the handshake protocol—for TCP-based connections the server opens with the handshake after accepting the connection—but uses the same message types. All subsequent BASP messages are processed independent of the transport protocol.

Adding UDP capabilities to the underlying components requires: (1) low-level event handlers responsible for reading and writing datagrams, (2) high-level broker servants that manage their respective low-level component and exchange data with the multiplexer, and (3) BASP broker support. The components that handle communication in the network stack are displayed in Table 6.1. Due to their similarity, the UDP components use the same names with a “`dgram_`” prefix.

6.3.1 Datagram Event Handlers

Low-level event handlers abstract over socket operations. They receive data on local sockets and sent data written into their buffers to the network. Callbacks into their managing brokers servants propagate events and data. The two new event handlers listed for UDP in Table 6.1 were added to the default multiplexer to handle datagram operations.

A datagram acceptor is registered for read events of its assigned socket and keeps a byte buffer for incoming datagrams. When called by the event loop, the acceptor reads the datagram into the buffer and calls the `new_endpoint` function of its parent broker servant. The servant accepts the receive buffer and reads address information of the sender from the acceptor.

Listing 6.3 shows processing of a new read event by the datagram acceptor. The listing omits error handling to focus on its functionality. Private member variables marked with the suffix ‘_’. The function `handle_event` is implemented by each event handler. Its argument

```
 1  void dgram_acceptor::handle_event(operation op) {
 2    if (mgr_ && op == operation::read) {
 3      if (!receive_datagram(bytes_read_, fd(), rd_buf_.data(),
 4                            rd_buf_.size(), sa_, sa_len_))
 5          // handle error
 6      if (bytes_read_ > 0) {
 7        std::tie(host_,port_) = sender_from_sockaddr(sa_, sa_len_);
 8        if (!mgr_->new_endpoint(rd_buf_.data(), bytes_read_))
 9          // handle error
10      }
11      prepare_next_read();
12    }
13  }
```

Listing 6.3: A datagram acceptor receives a datagram.

signifies the operation to process the event. The datagram acceptor only handles read events and ignores other types of events. First, the function checks if the acceptor has a manager assigned (accessed through the `mgr_` member) and was called to handle a read event (line 2). Then, it calls the function `receive_datagram` which performs the call to `recvfrom` of the socket API. The function accepts an integer per reference as a first argument which is filled with the amount of bytes read. The second argument is the socket file descriptor for receiving, followed by a receive buffer and its size. The last two arguments are a `sockaddr_storage` and its size which are used to store the address of the sender. If no error occurred and payload was received—datagrams of size zero are valid but can be ignored here—the acceptor parses host and port information from the `sockaddr_storage` before calling its manager to process the new information (line 8). Finally, the acceptor prepares the next read event (line 11) which ensures that the receive buffer is sized correctly.

A datagram stream is registered for read as well as write events. Similar to the acceptor, it keeps a byte buffer for incoming datagrams. In addition, a queue with byte buffers holds outgoing messages, each sent as a single datagram. Message buffers appended to queue should adhere to network and datagram limits since the stream does not perform slicing.

The function `handle_event` of the datagram stream uses a switch statement to different between read and write operations. Listing 6.4 shows how read events are processed while Listing 6.5 depicts the code for write events. Read events are processed similarly in acceptors and streams. First, the stream calls the function `receive_datagram` to read the datagram into a local buffer. It passes a local variable to store the number of received bytes as a first argument the socket file descriptor as a second argument. Followed by the receive buffer and its size. The last two arguments are a `sockaddr_storage` and its length to accept the

```
 1 case operation::read: {
 2   if (!receive_datagram(bytes_read_, fd(), rd_buf_.data(),
 3                         rd_buf_.size(), sa_, sa_len_))
 4     // handle error
 5   if (bytes_read_ > 0) {
 6     std::tie(host_, port_) = sender_from_sockaddr(sa_, sa_len_);
 7     if (!reader_->consume(&backend(), rd_buf_.data(), bytes_read_))
 8       // handle error
 9   }
10   prepare_next_read();
11   break;
12 }
```

Listing 6.4: A datagram stream received a datagram.

address information of the sender. If the datagram was received successfully and contained payload, the stream parses the information of the sender (line 6) and propagate the event to its managing broker servant (`reader_`) via the function `consume` (line 7). It expects a pointer to the multiplexer, a pointer to the received data and the number of received bytes. Finally, the stream prepares for the next operation (line 10) which prepares a the receive buffer for the next read event.

Listing 6.5 shows how read events are handled by a datagram stream. In addition to the queue with outgoing messages, a stream keeps a send buffer that contains the next outgoing message (`wr_buf_`). The function `send_datagram` wraps the `sendto` call of the socket API (line 3). It accepts a reference to an integer as a first arguments that will be assigned with the number of written bytes. The second argument is the file descriptor of the sending socket. The next two arguments are the send buffer the number of bytes it contains. The last two arguments indicate the receiver stored in a `sockaddr_stroage` and its size. If the datagram was sent successfully and the stream was configured to notify its manager (`writer_`) about send events, it calls function `datagram_sent`. Finally, the stream prepares the next write event (line 8). This operation moves the head of the queue into the send buffer (`wr_buf_`) or deregisters the datagram stream from receiving write events—until new data is appended to its queue.

UDP event handlers require configuration of the sizes of their receive buffers. A datagram is received in single `recvfrom` call, discarding excess bytes that do not fit into the buffer. They offer a member function `configure_datagram_size` to allow configuration of the their buffers.

```
case operation::write: {
  size_t wb; // written bytes
  if (!send_datagram(wb, fd(), wr_buf_.data(), wr_buf_.size(),
                     remote_addr_, remote_addr_len_))
    // handle error
  if (ack_writes_)
    writer_->datagram_sent(&backend(), wb);
  prepare_next_write();
  break;
}
```

Listing 6.5: A datagram stream sends a datagram.

6.3.2 Datagram Broker Servants

Brokers interact with the event handlers through broker servants. The servants implemented for UDP are `dgram_doorman` and `dgram_scribe` listed in Table 6.1. In addition to callbacks to propagate errors, the servants offer the callbacks used by the datagram acceptor and stream in the listings above.

While a doorman that accepted a TCP connection can initialize a new scribe with the socket handle returned from `accept`, there is no similar function for UDP. Instead, a datagram acceptor calls the function `new_endpoint` of its managing broker servant, depicted in Listing 6.6. The function creates a new datagram scribe responsible for communication with the new endpoint before forwarding the event to the BASP broker. It accepts a pointer to a buffer with the received datagram and size of the buffer as arguments. The `acceptor_` member is the datagram acceptor that called the function. It offers access to the address information of the last sender through the function `last_sender` and access to the multiplexer via the function `backend` which is used to create new broker servants.

```
1  bool new_endpoint(const void* buf, size_t num_bytes) {
2    auto& dm = acceptor_.backend();
3    auto endpoint_info = acceptor_.last_sender();
4    auto fd = new_dgram_scribe_impl(endpoint_info);
5    auto hdl = dm.add_dgram_scribe(parent(), fd, endpoint_info);
6    return dgram_doorman::new_endpoint(&dm, hdl, buf, num_bytes);
7  }
```

Listing 6.6: The `new_endpoint` callback of the datagram doorman.

First, a new socket is created via the function `new_dgram_scribe_impl` (line 4) which creates a socket for a datagram scribe and binds it to an open port. The function accepts endpoint information to create a socket matching the protocol version. Next, a new datagram scribe is created via the function `add_dgram_scribe` of the multiplexer (line 5). It accepts a pointer to the managing broker (acquired by `parent()`), a socket and information of the endpoint the scribe is responsible for. The function returns a handle to the newly created datagram scribe. Lastly, the function `new_endpoint` of the base class is called. It packs the arguments in a message and sends it to the parent broker.

The message sent to the broker is of type `new_endpoint_msg`, see the first struct in Listing 6.7. In addition to a handle that identifies the datagram doorman that sent the message (`source`), the message contains the received datagram (`buf`), a handle to the scribe responsible for the new endpoint (`handle`) as well as the port of the socket that received the datagram—which is used to look up the published actor.

Datagram scribes offer the function `consume` called by datagram streams after receiving a message. The consume function packs the received message in a message of type `new_datagram_msg`, the second message type listed in Listing 6.7 and sends it to the broker.

```
/// Signalizes newly discovered remote endpoint to a broker.
struct new_endpoint_msg {
  // Handle to the datagram endpoint that sent the message.
  dgram_doorman_handle source;
  // Buffer containing the received data.
  std::vector<char> buf;
  // Handle to new datagram scribe responsible for the communication
  dgram_scribe_handle handle;
  // Port of the addressed actor
  uint16_t port;
};

/// Signalizes that a datagram with a certain size has been sent.
struct new_datagram_msg {
  // Handle to the endpoint that received the data.
  dgram_scribe_handle handle;
  // Buffer containing received data.
  std::vector<char> buf;
};
```

Listing 6.7: Datagram related message types handled by brokers.

6.3.3 Datagram Processing in the BASP Broker

Brokers provide an actor-based abstraction over network I/O. Broker servants—such as the doorman and scribe—notify their parent broker about received data and other network events by sending messages. The BASP broker is a stateful actor, i.e., a function-based actor that keeps an object of a specified type as its state. The class `basp_broker` is derived from the class `broker` which in turn inherits from the class `abstract_broker` (shortly discussed in 5.5).

In conjunction with new UDP-related event handlers and broker servants, the BASP broker was extended to work with these components. This includes processing of the new message types shown in Listing 6.7 which require separate processing from the respective TCP-related messages, `new_connection_msg` and `new_data_msg`, to address the different communication semantics. The introduction of new servant types required small changes to the management structure of the BASP broker and its general processing steps which enable management of different types of servant handles.

When receiving data, the handlers have to differentiate between the continuous byte buffer offered by TCP and the datagrams for UDP. Listing 6.8 shows how both messages are handled by the BASP broker. The first message handler (lines 2 to 16) processes a stream of bytes and the second one (lines 18 to 24) processes a stream of datagrams. The signature of each lambda is matched against types in incoming messages. Pattern matching stops on the first match. In this case, both handlers match a message that contains a single object of type `new_data_msg` or `new_datagram_msg`, respectively. Code comments in the listing signify omitted code to focus on the message processing.

First, each message handler queries context information (lines 3 to 4 and 19 to 20) for processing such as the header of the current BASP message or state to provide guarantees for datagrams. Subsequently, both handlers call the function `handle` of the local BASP instance (lines 5 and 21), passing the received message as well as the header saved in the context.

The receive buffer offered by TCP contains a continuous stream of bytes that requires slicing into individual messages. A complete BASP message is acquired by handling two data messages: one for the BASP header to parse the size of the payload, and one for the payload. For this purpose, the message handler is called twice, changing the size of the next data chuck (lines 10 to 12) either to the payload length (line 11) or the BASP header size (line 12). This information is passed to the scribe (line 13) before saving the connection state for the next message. The additional boolean passed to handle (line 6) signifies what type of message is expected, header or payload. In contrast, a `new_datagram_msg` (lines 18 to 24) always contains a complete BASP messages that can be split into header and payload by during processing in the function `handle` (line 21).

```
// Handles received data from TCP-based scribes
[=](new_data_msg& msg) {
  state.set_context(msg.handle);
  auto& ctx = *state.this_context;
  auto next = state.instance.handle(msg, ctx.hdr,
                                    ctx.cstate == basp::await_payload);
  if (next == basp::close_connection) {
    // close connection
  } else if (next != ctx.cstate) {
    auto rd_size = next == basp::await_payload
                 ? ctx.hdr.payload_len
                 : basp::header_size;
    configure_read(msg.handle, receive_policy::exactly(rd_size));
    ctx.cstate = next;
  }
},
// Handles received data from UDP-based scribes
[=](new_datagram_msg& msg) {
  state.set_context(msg.handle);
  auto& ctx = *state.this_context;
  auto err = state.instance.handle(msg, ctx.hdr);
  if (err)
    // handle error
}
```

Listing 6.8: The BASP broker handles received data.

The BASP broker has to adhere to the MTU limitations for BASP messages it sends via datagrams and slice messages accordingly. It packs each slice in a separate BASP messages, marking and numbering them to allow assembly at the receiver side. Note, that slicing is not implemented yet, but will be added before the code is merged into the stable code base.

Listing 6.9 shows the handlers for the new connection and new endpoint messages. The first message handler (lines 2 to 9) handles a newly accepted TCP connection. It acquires a reference to the local BASP instance (line 3) and sends a handshake to the new remote endpoint. This is accomplished with the BASP function `write_server_handshake` (line 4), passing the buffer of the responsible scribe and the port of the doorman. The port information allows the runtime environment to lookup information of an actor published on the port. The following call to flush (line 6) ensures that data passed to a scribe such as the handshake written into its buffer is sent. Finally, the chunk size to deliver in the next `new_data_msg` is set to the BASP header size (line 7).

```
1  // A new TCP connection
2  [=](const new_connection_msg& msg) {
3    auto& bi = state.instance;
4    bi.write_server_handshake(wr_buf(msg.handle),
5                              local_port(msg.source));
6    flush(msg.handle);
7    configure_read(msg.handle,
8                   receive_policy::exactly(basp::header_size));
9  },
10 // A new UDP endpoint
11 [=](new_endpoint_msg& msg) {
12   state.set_context(msg.handle);
13   auto& ctx = *state.this_context;
14   auto err = state.instance.handle(msg, ctx.hdr);
15   if (err)
16     // handle error
17   configure_datagram_size(msg.handle, dgram_buf_size);
18 }
```

Listing 6.9: The BASP broker handles contact from remote nodes.

Whenever a datagram doorman receives a message from an unknown endpoint, it sends a `new_enpoint_msg` to the broker which includes the received datagram. The message handler (lines 11 to 18) has to process the message similarly to a normal datagram. For this purpose, the handler acquires a reference to the context for communication handled by the newly created scribe (lines 12 to 13). Thereafter, it calls the function `handle` of the local BASP instance (line 14). If the message contains the expected client handshake, the server handshake will be written into the the buffer of the datagram scribe as an answer. Finally, the message handler configures the receive buffer for the next datagram (line 17).

6.4 Discussion

The implementation presented here enables UDP communication between distributed CAF nodes. It is strongly influenced by the existing TCP implementation and follows a similar design. For this purpose, the application handshake between BASP brokers was adapted to work without the TCP connection. Moreover, the BASP broker was extended to parse datagrams into BASP messages. Datagrams are processed at the multiplexer using datagram doormen to accept requests from new endpoints and datagram scribes for regular communication. A newly introduced URI class allows developers to specify host and port information using the scheme field to choose the transport protocol. Supported schemes are currently `tpc` and `udp`. With basic UDP support available, the next steps concern the following topics, discussed in Section 5.5:

Generalization The implementation presented here works for communication between CAF nodes. However, brokers should offer a generalized abstraction over the network and give developers the opportunity to implement network services that offer an actor interface internally. The behavior of the UDP communication may be unexpected for a generic interface, e.g., the socket and port for datagram answers may differ from the port that accepted the request.

Guarantees Brokers can now exchange messages with remote nodes using raw UDP. For communication between BASP brokers messaging should satisfy the set of guarantees set in Section 5.4 as part of the design. Moreover, ordering is a key requirement to *slice* large messages into smaller datagrams. State in form of message buffers and sequence numbers can be kept in the endpoint specific context kept by the BASP broker, while processing is handled in the protocol specific `handle` function before the message is evaluated as a generic BASP message.

Ordering requires only local processing of sequence numbers included in each message. Messages that arrive out of order are buffered until all messages with a smaller sequence number are delivered. In the presence of unreliable messaging, the broker buffers messages for a limited time and discards messages that arrive late. Timeouts are realized as delayed the broker sends to itself. On receipt it delivers the message with the sequence number from message if that did not happen in the meantime.

Reliable delivery is more complicated as it requires communication with remote nodes. There are many protocols offering reliable message transfer for example TCP [80] implementing cumulative and selective acknowledgements or CoAP [67] which addresses constrained environments. A simple approach resends messages if it did not receive an acknowledgement within a certain time frame. Acknowledgement can be bundled instead of acknowledging each receipt individually. In any case, the algorithms should be configurable to address different deployment scenarios. After reliability is available, the send API requires the option to flag messages as reliable, as discussed in Section 5.3.

Monitoring remote nodes is required for error detection and propagation. UDP does not have a connection state that can be observed for such purposes. Hence, a general failure detector implemented for UDP could monitor remote nodes independent of the protocol in use. Suitable failure detectors were discussed in Section 4.2.3.

Routing Table The routing table keeps mappings between node ids and handles to use for the communication. Capabilities for multi-hop routing have already been remove from the table. However, no replacement for dynamic addressing has been added yet. The routing functionality is generally independent of datagram communication and can be implemented orthogonally. Without message routing, the node id field can be removed from the BASP header as messages are only exchange between neighbors and always address the receiver.

Protocol Policies Policies were discussed as a concept in Section 5.5. Before starting the implementation, the datagram specific components of the middleman and multiplexer should be stable. Similar to routing changes, this implementation is a separate task—although partially dependent on the datagram, it can be released separately.

After adding new features to the network stack, performance measurements should confirm that the additional features do not impact performance. With the introduction of a new transport protocol, its performance should be compared to the previously existing implementation. Especially, the influence of application layer guarantees on datagrams should be measured and compared to TCP which already includes such functionality.

The final step before merging the code is an update to the manual explaining (1) the guarantees CAF offers for message passing, (2) how UDP can be used as a transport protocol in CAF and (3) what steps are required to implement additional transport protocols.

7 Evaluation

CAF implements unit tests for most of its functionality in order to provide a robust and reliable framework. The library for testing is shipped with the framework itself and called `libcaf_test`. The default build configuration includes core, io and test libraries as well as the unit test. After building the library, the command "`make test`" runs all unit tests.

Functionality was added to the BASP broker, middleman and multiplexer during the implementation. The existing unit tests ensure that previous functionality was not broken. Extending the unit tests to include datagram related functionality ensures proper algorithm functionality. This chapter shortly summarizes testing of the implementation and discusses the limits of unit tests.

7.1 Unit Tests

Brokers depend on the multiplexer interface for network interactions. Per default, CAF deploys a multiplexer of class `default_multiplexer` for this purpose. The multiplexer can be exchanged with an implementation called `test_multiplexer` which offers a controlled environment for testing. This multiplexer neither implements any network operations nor does it have its own event loop. Instead, the test environment has control over processing in the multiplexer and can examine its state in each step.

The test multiplexer includes broker servants that draw their data from state kept by the multiplexer instead of from event handlers. Messages can be appended directly since servants expose their buffers. A call to the related read function lets the servant process the message, e.g., by forwarding the buffer to its parent broker wrapped in the respective message. In the same way, messages sent by the broker can be read from the buffers and tested for correctness. For UDP related testing the test multiplexer was extended with the related servant classes `dgram_scribe` and `dgram_doorman`.

CAF implements a test suit for the BASP broker that validates its messaging behavior such as handshake or delegate messages and tests the interaction of the broker with its servants. The cases related to the TCP related messages and servants were extended to test the UDP functionality as well. Listing 7.1 depicts a test case that publishes an actor and established communication with it. The CAF_TEST macro in the first line bundles a single test and provides a fresh fixture which provides access to the test multiplexer using the function mpx, keeps a local actor accessible through self, and offers a simulated remote node through the function jupiter.

The test case publish_and_connect_udp first generates a datagram doorman handle (line 2) and creates an associated doorman for the handle in the test multiplexer, using port 4242. Subsequently, the test creates a URI, passing 0.0.0.0 as an IP allows contact from any remote address, to publish the self actor of the fixture. Publish asks the test multiplexer for a doorman that handles the given URI, in this case acquiring the datagram doorman created in line 3. The return value is the port of the published actor (4242). Due to the absence of the event loop, the function flush_runnables moves processing in the broker forward, causing the broker to process the publish message sent by the middleman. Finally, the function connect_node accepts a simulated node (jupiter), a doorman handle (ax) and the id of a published actor (self) to perform and verify the handshake process.

```
1  CAF_TEST(publish_and_connect_udp) {
2    auto ax = dgram_doorman_handle::from_int(4242);
3    mpx()->provide_dgram_doorman(4242, ax);
4    auto u = uri::make("udp://0.0.0.0:4242");
5    CAF_REQUIRE(u);
6    auto res = system.middleman().publish(self(), *u);
7    CAF_REQUIRE(res == 4242);
8    mpx()->flush_runnables(); // processing in the basp_broker
9    connect_node(jupiter(), ax, self()->id());
10 }
```

Listing 7.1: Testing UDP related functionality of the BASP broker.

Checks are performed with macro CAF_REQUIRE which accepts a boolean expression and throws an exception if the expression does not evaluate to true. The macro CAF_CHECK is a weaker guard that only reports expressions that evaluate to false, but does not interrupt the test.

Testing the network capabilities of the default multiplexer requires separate testing. The unit tests include tests for the remote actor functionality. CAF bundles components such as the scheduler, BASP broker middleman and multiplexer in a class called actor_system. Creating multiple actor systems in a single process allows communication using local sockets. Although timing-dependent network characteristics such as delay or jitter cannot be tested in this way, the general functionality of remote communication using actor messages can be evaluated.

Especially for performance and efficiency, distributed measurements are necessary to confirm functionality and scalability.

7.2 Testing Functionality

In addition to the unit tests, a small application scenario was implemented to confirm functionality of the datagram implementation. It tests the capability of the multiplexer to multiplex communication with multiple UDP and TCP endpoints.

The test application creates a single actor and publishes it on a UDP port and a TCP port. Since the behavior for the actor is independent of the multiplexer logic and thus not important for this test, it only performs a simple task and answers to incoming integers with the incremented input. Client nodes accept a URI to acquire a proxy of the server actor and exchange an integer with the server until a limit is reached. For the test setup, 20 clients sent requests to a single server in parallel using a mix of TCP and UDP for communication. The clients were hosted separately form the server relying on a wireless network for communication. As a result, the server handled the communication without interruptions showing the expected behavior for multiplexing socket communication.

A second test scenario provokes message loss, a behavior that is expected from pure UDP communication and should be addressed in future work. The setup consists of two CAF nodes, each in a separate process, that communicate via a local socket.

The server application creates an actor that accepts regular messages consisting of an integer and a vector of configurable size. The integer works as a sequence number to check message loss or out of order delivery. The vector simulates payload. In addition, it reacts to a quit message by printing the number of received messages and exiting. The client application acquires a handle of the published server actor and passes it to a local actor that sends a configurable amount of messages containing a sequence number and a payload to the server before sending a quit message. It accepts configuration for the number of messages, the payload size and the interval between each message in microseconds. Both applications accept configuration of the transport protocol.

Message loss can be observed on a local link, sending a total of 10^5 messages with a payload of 1024 bytes every microsecond leads to message loss. The amount of message loss varies between 0% and 2%. Using a wireless network for the communication raises message loss to more than 90% for the same configuration. Messages are lost because the bandwidth of the link is exceeded and messages are dropped after buffers run full. As expected, using TCP for the transport reduces message loss to 0% in both cases. Consistent with these observations, the using UDP requires only two thirds of the time to send its messages over the wireless network compared to TCP.

7.3 Discussion

The unit tests implemented in CAF allow testing the functionality of the BASP broker using an alternative multiplexer implementation. This test multiplexer was extended to test the UDP functionality of the BASP broker. After implementing changes to the overlay routing of CAF, the BASP unit tests should be extended to test this functionality. In addition, timing influences, sharing of address information and reactive establishment of communication between nodes requires testing. A small test setup confirmed functionality of socket multiplexing in a mixed setup of UDP and TCP.

Operational functionality of messaging guarantees for datagram communication can be tested only to limited extend in unit tests. For ordering and message loss, the multiplexer can be extended with configuration options to drop messages or deliver them out of order. While this ensures that the related algorithms work correctly, local testing does not provide information about performance or real world application. Even tests between local nodes do not provide a representative scenario as Internet-wide communication or IoT scenarios exhibit different characteristics. Furthermore, monitoring guarantees depend on timing which is inherently differs greatly between a local node and a distributed system. Networks can be simulated to achieve a more realistic scenario.

In addition to viability in different deployment environments, the efficiency and performance of individual guarantees are of interest. Throughput, message overhead and number of control messages are relevant and should be compared to the raw protocol usage as well as TCP. Moreover, general system load and resource consumption should be evaluated.

8 Conclusion & Outlook

The network layer used today by the C++ Actor Framework (CAF) for communication in distributed systems largely grew with its challenges. It enables the management of distributed CAF nodes and inherits its messaging guarantees from the TCP transport protocol. Similar to other actor systems, the messaging guarantees between CAF actors differ under distribution from local concurrency. For example, message passing to remote actors is often unreliable while messages are reliably delivered to local actors.

This thesis examined how to address these discrepancies and how to improve the CAF distribution layer in the dimensions transparency, efficiency and robustness. For this purpose, the requirements for inter-actor communication were evaluated. Central considerations were the reliability aspects of message passing with regard to message delivery, ordering and monitoring. Each aspect was evaluated with respect to efficiency, adaptive flexibility and state requirements to assess its impact on runtime behavior and define the guarantees for message passing in CAF.

Although delivery is reliable on a local node, the new design defines message delivery to be unreliable per default. This enables light-weight message passing with minimal overhead. Complementary guarantees are still added by the underlying transport protocol. Optional reliable delivery decouples delivery guarantees from the transport layer and addresses a frequently desired use-case.

Messages are ordered causally on local nodes. This is a strong guarantee and expensive to establish in a distributed system where it requires synchronization, additional information in each message or a controlled topology. It impacts performance and scalability enough to refrain from causal ordering for actor message passing and fall back to FIFO ordering.

Actors reliably receive error messages when a linked or monitored actor fails. The network introduces new types of failures for unreachable nodes due to link or node failure. Complex applications that manage highly distributed actor systems require knowledge of these failure types to handle failure recovery. As a result, error handling differs for local and remote failures.

The tight coupling to a specific transport protocol was revisited in conjunction with the reliability considerations. The choice of transport protocols is relevant in many dimensions and can provide environment-specific functionality such as connectivity in restricted networks or reduced latency. Offering an extensible network layer enables developers to integrate new transport protocols and adapt to future deployment scenarios. Late binding of transport protocols allows dynamic deployment and enables developers to write flexible software. Establishing a consistent set of guarantees in the presence of exchangeable transport protocols enables predictable behavior as the guarantees provided by different protocols may differ greatly.

The overlay network build between distributed CAF nodes allowed message routing to establish transparent message passing. To reduce complexity and strengthen the guarantees offered by transport protocols, the routing capabilities are removed from the overlay. Instead, the runtime environment relies on the underlay to deliver messages between each pair of CAF nodes.

With all these considerations in mind, a design for the network stack was proposed. It introduces URIs as a meta-type to specify endpoints using the scheme field to specify the transport protocol. Additional transport protocols can be added in one of two ways. First, protocols that require deep integration require the implementation components to manage protocol-specific socket and integration into CAF brokers to handle related transmission characteristics. Second, a policy-based design allows the extension of integrated protocols in an easier way. The BASP broker strengthens guarantees on the application layer for transport protocols that do not provide them by design. Available protocols can be used simultaneously for different endpoints or be chosen to match the deployment needs. Finally, the overlay routing functionality is removed from CAF in favor of reactively established direct communication between nodes. Thus, transferring routing responsibilities from CAF to the underlay network.

An implementation that integrates UDP transport into the multiplexer, middleman and BASP broker was presented. It enables raw datagram transmission without guarantees and paves the way for additional features such as ordering, reliable delivery and message slicing. A basic evaluation extended CAF unit tests to cover datagram functionality and ensured continued functionality of the multiplexer and its capabilities to handle UDP and TCP endpoints simultaneously.

While this thesis discussed conceptual work extensively, the implementation of the proposed design is not yet finished. The following list provides an overview over the next steps towards a complete implementation:

Datagrams The multiplexer and BASP broker were adapted to handle datagram communication. For now, CAF does not necessarily use the same socket and port for datagram exchange with a remote endpoint. This might not be expected by remote endpoints which could use address and port information to identify endpoints. A generalization of the implementation eases UDP-based communication between brokers and generic UDP endpoints.

Guarantees The guarantees for message passing were defined in the design goals. The BSAP broker is responsible to uphold these guarantees for actor messages independent of the transport protocol. The implementation presented in Chapter 6 only offers raw UDP. In the next steps, ordering, reliability, slicing and monitoring guarantees will be added to the BASP broker. The implementation should be general enough to allow future transport protocols to adapt the guarantees easily. In conjunction with each guarantee, the capabilities of the test multiplexer will be enhanced to allow proper unit testing.

Addressing Removing routing from the CAF overlay removes the routing table from the framework. Instead nodes exchange addresses and reactively establish communication when they encounter actors from remote nodes. In this context, the structure of actor identifiers should be reconsidered. A separation of locator and identifier for nodes in actor systems is not necessary since actors are fixed to a node. This allows encoding of location into their identifiers. Moreover, implications on performance and efficiency should be carefully considered since actor handles are accessed and shared frequently.

Protocol Policies The interface for protocol policies can be added to CAF once the datagram implementation is complete. The general concept was discussed in Section 5.5. An implementation of transport protocols as policies is required for validation. HTTP tunneling is considered as a first implementation for the TCP-based policies. CoAP is a candidate for a UDP-based policies and was already considered for CAF messaging in IoT environments [15].

Benchmarking The tests discussed in Chapter 7 are a fist steps to evaluate the datagram communication in CAF. With the addition of guarantees, addressing changes and protocol policies additional unit test will be added. Going a step further, the performance of each component should be measured and assessed. For this purpose, each guarantee will be measured and compared to the raw protocol as well as to TCP, which already implements each one. Furthermore, the evaluation should examine how the reworked addressing affects functionality and scalability in form of message exchanges and state size.

Documentation This work provided a definition for the messaging guarantees of actor messages in CAF. Since this affects application development, the manual should communicate this accordingly. The process to include new transport protocols—as a policy or deeply integrated—requires explanation and documentation to ease the process for interested developers.

Bibliography

[1] 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: ACM, Oct. 2013, pp. 87–96.

[2] 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: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

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

[4] G. Agha, I. A. Mason, S. Smith, and C. Talcott, "Towards a Theory of Actor Computation," in *Proc. of CONCUR*, ser. LNCS, vol. 630. Heidelberg: Springer-Verlag, 1992, pp. 565–579.

[5] 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: ACM, 2007, pp. 6–1–6–26.

[6] Typesafe Inc., "Akka Framework," http://akka.io, August 2017.

[7] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Revisiting Actor Programming in C++," *Computer Languages, Systems & Structures*, vol. 45, pp. 105–131, April 2016. [Online]. Available: http://dx.doi.org/10.1016/j.cl.2016.01.002

[8] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

[9] D. Charousset, R. Hiesgen, and T. C. Schmidt, "CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications," in *Proc. of the 5th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '14), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2014, pp. 15–28.

[10] T. Desell and C. A. Varela, “SALSA Lite: A Hash-Based Actor Runtime for Efficient Local Concurrency,” in *Concurrent Objects and Beyond*, ser. Lecture Notes in Computer Science, G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, Eds., vol. 8665. Springer Berlin Heidelberg, 2014, pp. 144–166.

[11] S. Srinivasan and A. Mycroft, “Kilim: Isolation-Typed Actors for Java,” in *Proc. of the 22nd ECOOP*, ser. LNCS, vol. 5142. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 104–128.

[12] L. V. Kale and S. Krishnan, “Charm++: Parallel programming with message-driven objects,” *Parallel Programming using C++*, pp. 175–213, 1996.

[13] M. Vallentin, V. Paxson, and R. Sommer, “VAST: A Unified Platform for Interactive Network Forensics,” in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2016.

[14] M. Vallentin, D. Charousset, T. C. Schmidt, V. Paxson, and M. Wählisch, “Native Actors: How to Scale Network Forensics,” in *Proc. of ACM SIGCOMM, Demo Session*. New York: ACM, August 2014, pp. 141–142.

[15] R. Hiesgen, D. Charousset, and T. C. Schmidt, “Embedded Actors – Towards Distributed Programming in the IoT,” in *Proc. of the 4th IEEE Int. Conf. on Consumer Electronics - Berlin*, ser. ICCE-Berlin’14. Piscataway, NJ, USA: IEEE Press, Sep. 2014, pp. 371–375.

[16] R. Hiesgen, D. Charousset, T. C. Schmidt, and M. Wählisch, “Programming Actors for the Internet of Things,” *Ercim News*, vol. 101, pp. 25–26, April 2015. [Online]. Available: http://ercim-news.ercim.eu/en101/special/programming-actors-for-the-internet-of-things

[17] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, “RIOT OS: Towards an OS for the Internet of Things,” in *Proc. of the 32nd IEEE INFOCOM. Poster.* Piscataway, NJ, USA: IEEE Press, 2013.

[18] R. Hiesgen, D. Charousset, and T. C. Schmidt, “Manyfold Actors: Extending the C++ Actor Framework to Heterogeneous Many-Core Machines using OpenCL,” in *Proc. of the 6th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH ’15), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2015, pp. 45–56.

[19] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” IETF, RFC 5246, August 2008.

[20] E. Rescorla and N. Modadugu, “Datagram Transport Layer Security,” IETF, RFC 4347, April 2006.

[21] Lightbend Inc., *Akka Scala Documentation Release 2.4.1*, Nov 2015, http://doc.akka.io/docs/akka/current/AkkaScala.pdf, Accessed: 09-02-2016.

[22] J. Armstrong, "Making Reliable Distributed Systems in the Presence of Software Errors," Ph.D. dissertation, Department of Microelectronics and Information Technology, KTH, Sweden, 2003.

[23] H. Svensson and L.-Å. Fredlund, "Programming Distributed Erlang Applications: Pitfalls and Recipes," in *Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop*, ser. ERLANG '07. New York, NY, USA: ACM, 2007, pp. 37–42.

[24] P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, "Orleans: Distributed Virtual Actors for Programmability and Scalability," Microsoft, Tech. Rep. MSR-TR-2014-41, March 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=210931

[25] M. de Graauw, "Nobody Needs Reliable Messaging," http://www.infoq.com/articles/no-reliable-messaging, Jun 2010, accessed: 08-02-2016.

[26] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-End Arguments in System Design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, Nov 1984.

[27] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[28] F. Mattern, "Virtual Time and Global States of Distributed Systems," in *Parallel and Distributed Algorithms*. North-Holland, 1989, pp. 215–226.

[29] K. Birman, A. Schiper, and P. Stephenson, "Lightweight Causal and Atomic Group Multicast," *ACM Trans. Comput. Syst.*, vol. 9, no. 3, pp. 272–314, Aug. 1991.

[30] A. Schiper, J. Eggli, and A. Sandoz, *A new algorithm to implement causal ordering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 219–232.

[31] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, "Fast Message Ordering and Membership Using a Logical Token-passing Ring," in *13th Int. Conf. on Distributed Computing Systems*, ser. ICDCS '93. Washington, DC, USA: IEEE Computer Society, 1993, pp. 551–560.

[32] Y. Long, M. Bagherzadeh, E. Lin, G. Upadhyaya, and H. Rajan, “On Ordering Problems in Message Passing Software,” in *Modularity’16: 15th International Conference on Modularity*, ser. Modularity’16, March 2016. [Online]. Available: http://design.cs.iastate.edu/papers/MODULARITY16a

[33] S. Blessing, “A String of Ponies,” http://www.doc.ic.ac.uk/teaching/distinguished-projects/2013/s.blessing.pdf, Imperial College London, MSc Thesis, Sep. 2013.

[34] P. A. S. Ward, “Algorithms for Causal Message Ordering in Distributed Systems.” [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.1137&rep=rep1&type=pdf

[35] B. Charron-Bost, “Concerning the Size of Logical Clocks in Distributed Systems,” *Inf. Process. Lett.*, vol. 39, no. 1, pp. 11–16, Jul. 1991. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(91)90055-M

[36] F. Adelstein and M. Singhal, “Real-time Causal Message Ordering in Multimedia Systems,” in *Proceedings of the 15th International Conference on Distributed Computing Systems*, ser. ICDCS ’95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 36–43. [Online]. Available: http://dl.acm.org/citation.cfm?id=876885.880042

[37] S. C. Kendall, J. Waldo, A. Wollrath, and G. Wyant, “A Note on Distributed Computing,” Mountain View, CA, USA, Tech. Rep., 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=974938

[38] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of Distributed Consensus with One Faulty Process,” *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985. [Online]. Available: http://doi.acm.org/10.1145/3149.214121

[39] T. D. Chandra and S. Toueg, “Unreliable Failure Detectors for Reliable Distributed Systems,” *J. ACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996. [Online]. Available: http://doi.acm.org/10.1145/226643.226647

[40] M. Bertier, O. Marin, and P. Sens, “Implementation and Performance Evaluation of an Adaptable Failure Detector,” in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, ser. DSN ’02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 354–363. [Online]. Available: http://dl.acm.org/citation.cfm?id=647883.738261

[41] 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*, ser. SRDS ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 66–78.

[42] W. Chen, S. Toueg, and M. K. Aguilera, “On the Quality of Service of Failure Detectors,” *IEEE Trans. Comput.*, vol. 51, no. 5, pp. 561–580, May 2002. [Online]. Available: http://dx.doi.org/10.1109/TC.2002.1004595

[43] C. Fetzer, M. Raynal, and F. Tronel, “An Adaptive Failure Detection Protocol,” in *Dependable Computing, 2001. Proceedings. 2001 Pacific Rim International Symposium on*, 2001, pp. 146–153.

[44] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer, “A Lazy Monitoring Approach for Heartbeat-Style Failure Detectors,” in *Third Int. Conf. on Availability, Reliability and Security*, ser. ARES ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 404–409.

[45] R. Braden, “Requirements for Internet Hosts - Communication Layers,” IETF, RFC 1122, October 1989.

[46] P. Mockapetris, “Domain names - concepts and facilities,” IETF, RFC 1034, November 1987.

[47] K. Pentikousis, B. Ohlman, D. Corujo, G. Boggia, G. Tyson, E. Davies, A. Molinaro, and S. Eum, “Information-Centric Networking: Baseline Scenarios,” IETF, RFC 7476, March 2015.

[48] T. Desell and C. A. Varela, “A Performance and Scalability Analysis of Actor Message Passing and Migration in SALSA Lite,” Tech. Rep., Oct. 2015, presented at the 5th International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!), held in conjunction with ACM SIGPLAN SPLASH. [Online]. Available: http://people.cs.und.edu/~tdesell/papers/2015_agere_salsa.pdf

[49] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, “Session Traversal Utilities for NAT (STUN),” IETF, RFC 5389, October 2008.

[50] P. Srisuresh, B. Ford, and D. Kegel, “State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs),” IETF, RFC 5128, March 2008.

[51] R. Mahy, P. Matthews, and J. Rosenberg, “Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN),” IETF, RFC 5766, April 2010.

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

[53] J. Rosenberg, A. Keranen, B. B. Lowekamp, and A. B. Roach, “TCP Candidates with Interactive Connectivity Establishment (ICE),” IETF, RFC 6544, March 2012.

[54] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, “SIP: Session Initiation Protocol,” IETF, RFC 3261, June 2002.

[55] M. Handley and V. Jacobson, “SDP: Session Description Protocol,” IETF, RFC 2327, April 1998.

[56] J. Hill, “Bypassing Firewalls: Tools and Techniques,” in *12th Annual FIRST Conference*, 2000.

[57] S. Kent and K. Seo, “Security Architecture for the Internet Protocol,” IETF, RFC 4301, December 2005.

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

[59] ——, “Transmission Control Protocol,” IETF, RFC 793, September 1981.

[60] R. Stewart, “Stream Control Transmission Protocol,” IETF, RFC 4960, September 2007.

[61] M. Tuexen and R. Stewart, “UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication,” IETF, RFC 6951, May 2013.

[62] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk, “QUIC: A UDP-Based Multiplexed and Secure Transport,” IETF, Internet-Draft – work in progress 01, October 2016.

[63] M. Thomson and R. Hamilton, “Using Transport Layer Security (TLS) to Secure QUIC,” IETF, Internet-Draft – work in progress 01, October 2016.

[64] R. Shade and M. Warres, “HTTP/2 Semantics Using The QUIC Transport Protocol,” IETF, Internet-Draft – work in progress 00, July 2016.

[65] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3,” IETF, Internet-Draft – work in progress 21, July 2017.

[66] E. Rescorla and N. Modadugu, “Datagram Transport Layer Security Version 1.2,” IETF, RFC 6347, January 2012.

[67] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," IETF, RFC 7252, June 2014.

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

[69] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," IETF, RFC 7540, May 2015.

[70] C. Holmberg, S. Hakansson, and G. Eriksson, "Web Real-Time Communication Use Cases and Requirements," IETF, RFC 7478, March 2015.

[71] H. Alvestrand, "Transports for WebRTC," IETF, Internet-Draft – work in progress 17, October 2016.

[72] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," IETF, RFC 7159, March 2014.

[73] C. Bormann and P. Hoffman, "Concise Binary Object Representation (CBOR)," IETF, RFC 7049, October 2013.

[74] Google, "Protocol buffers," https://developers.google.com/protocol-buffers/, accessed August 2016.

[75] N. Kushalnagar, G. Montenegro, and C. Schumacher, "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals," IETF, RFC 4919, August 2007.

[76] J. Hui and P. Thubert, "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks," IETF, RFC 6282, September 2011.

[77] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses," IETF, RFC 6824, January 2013.

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

[79] "Working Draft, Standard for Programming Language C++," International Organization for Standardization ISO/IEC, Geneva, CH, Working draft n4606, Jul 2016.

[80] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," IETF, RFC 2018, October 1996.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, November 13, 2017 Raphael Hiesgen