A Configurable Transport Layer for CAF

Raphael Hiesgen
Dept. Computer Science
HAW Hamburg
Germany
r.hiesgen@haw-hamburg.de

Dominik Charousset
Dept. Computer Science
HAW Hamburg
Germany
dcharousset@acm.org

Thomas C. Schmidt
Dept. Computer Science
HAW Hamburg
Germany
t.schmidt@haw-hamburg.de

Abstract

The message-driven nature of actors lays a foundation for developing scalable and distributed software. While the actor itself has been thoroughly modeled, the message passing layer lacks a common definition. Properties and guarantees of message exchange often shift with implementations and contexts. This adds complexity to the development process, limits portability, and removes transparency from distributed actor systems.

In this work, we examine actor communication, focusing on the implementation and runtime costs of reliable and ordered delivery. Both guarantees are often based on TCP for remote messaging, which mixes network transport with the semantics of messaging. However, the choice of transport may follow different constraints and is often governed by deployment. As a first step towards re-architecting actor-to-actor communication, we decouple the messaging guarantees from the transport protocol. We validate our approach by redesigning the network stack of the C++ Actor Framework (CAF) so that it allows to combine an arbitrary transport protocol with additional functions for remote messaging. An evaluation quantifies the cost of composability and the impact of individual layers on the entire stack.

CCS Concepts • **Networks → Programming interfaces**; *Transport protocols*; • **Computing methodologies → Distributed computing methodologies**;

Keywords Actor Model, Transport Layer, Networking, Reliability, Service Guarantees

ACM Reference Format:
Raphael Hiesgen, Dominik Charousset, and Thomas C. Schmidt. 2018. A Configurable Transport Layer for CAF. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE '18), November 5, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3281366.3281369

AGERE '18, November 5, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6066-1/18/11…$15.00
https://doi.org/10.1145/3281366.3281369

1 Introduction

Concurrency and distribution are an inherent part of modern systems and prevalent in most areas from personal computing and data centers to mobile platforms and the IoT. One challenge apparent in those areas is the dynamic adaption to the environment of deployment. Personal devices—often mobile notebooks, tablets, or phones—change locations, rely on cloud services, and regularly communicate through NATs and firewalls. For cloud scenarios and Mobile Edge Computing (MEC), this leads to service mobility and unpredictable location of nodes, which change application deployment according to user behavior. The IoT is still an emerging field with applications in home, infrastructure, and industrial automation targeted at a heterogeneous variety of deployments that include gateways.

The actor model of computation [15] seamlessly integrates concurrency and distribution, and gains popularity for designing and developing applications that meet the demands of flexible adaptivity and high scalability. Actors solely communicate via network-transparent message passing while applying a strong failure model. In reaction to a message, an actor can send messages, create new actors, or configure its future behavior. Actors offer a high level of abstraction that allows developers to focus on their application while the underlying framework takes responsibility for error prone tasks such as synchronization and networking—the implementations of which require domain-specific knowledge and experience.

Problem Statement Although the behavior of actors has been carefully modeled, their message passing layer lacks a clear definition. Communication guarantees regarding message delivery or ordering often diverge between implementations and contexts. For example, Armstrong [5] assumes message passing in Erlang "[. . .] to be unreliable with no guarantee of delivery". The Erlang software documentation closely couples reliability to the reliability of TCP transport. Similarly, according to its documentation[1] Akka delivers messages with an "at-most-once" semantics, even though

[1] https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html, accessed Aug'18

authors acknowledge that the guarantees are much stronger in local deployment. The same discrepancies can be found for ordering guarantees. Here non-local messages between a pair of actors often follow *FIFO* ordering while local messages are usually ordered *causally*—a result of synchronous calls to enqueue messages into a local mailbox.

There are many reasons for these discrepancies in the implementation of local and remote contexts [16]. First, local guarantees are much easier achieved than by protocols involving network communication, where uncertainty and unreliability need explicit treatment. Often an analysis of the alternatives and a reasoning for the offered guarantees is missing.

In practice, guarantees are often enforced by a tight coupling with the transport protocol of the desired characteristics. This approach may be acceptable for a large number of applications. However, it must be considered a severe limitation when scaling from small embedded devices over mobile and desktop to cloud services. 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 DCTCP or InfiniBand are optimized for closely coupled clusters. On the low end of the scale, constrained environments depend on specialized standards such as 6LoWPAN or CoAP over UDP to address a loose coupling in lossy networks.

Choosing a transport protocol is a trade-off between the scope of services a protocol offers and the environment of its operation. Simply deploying the protocol that offers the best guarantees is not a viable solution. While constrained environments might not be able to handle the messages sizes or network load, other applications may require low-latency and would rather loose messages than wait for retransmits. This trade-off further motivates the need for decoupling messaging guarantees from transport. Instead, an exchangeable transport layer augmented by configurable services can address scalability, adaptivity, as well as dynamic deployment at the same time.

Developers should be able to rely on a set of guarantees offered by a framework instead of rewriting applications to handle these tasks or tying communication to a specific protocol. These guarantees should be enforced across protocol choices and layers and allow transparent deployment of data transport based on the use case or environment.

Contributions In this work, we re-examine the duties and workings of actor communication with the goal to identify a set of reasonable guarantees for message passing between actors. The C++ Actor Framework [11, 12] is used as a reference. Specifically, we contribute:

1. A survey and discussion of communication aspects relevant to actor frameworks, focusing on reliable delivery and ordering.
2. A redesign of the CAF network layer to address our observations and allow a composable transport implementation.
3. A first evaluation of our design focusing on the cost of composability.

Overview Section § 2 discusses related work while Section § 3 introduces CAF, the framework hosting our subsequent work. § 4 reflects the main aspects considered in our design: reliable delivery and ordering. We present the redesign of the CAF network layer in § 5, and evaluate our implementation in § 6. Finally, § 7 concludes with an outlook.

2 Related Work

Reliable Delivery This aspect signifies how likely it is for a sent message to reach the destination and whether feedback is available in case of failure. Akka delivers messages unreliably with "at-most-once" semantics per default[2]. This means 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 [5]. The additional effort required to write applications that can handle unreliable message passing furthers scalability and increases robustness against errors. A later publication [6] goes into more detail on the topic and couples the reliability of messages passing to the reliability of TCP. However, TCP itself is not enough to guarantee delivery to an actor. Errors in the runtime environment (RE) can occur after a message 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. [24].

Microsoft released Orleans [7], 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 environment 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 a "maybe" delivery guarantee to avoid the associated costs in every message exchange. However, "at-least-once" delivery can be enabled, which retransmits messages until reception is acknowledged[3]. Since the RE does not detect

[2] https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html, accessed Aug'18

[3] http://dotnet.github.io/orleans/Documentation/clusters_and_clients/configuration_guide/messaging_delivery_guarantees.html, acc. Aug'18

duplicates, implementing "at-least-once" delivery burdens developers with deduplication in their implementation.

The blog post "Nobody Needs Reliable Messaging"[4] 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 et al. [22] 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 and reduce the complexity and overhead. 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 in regard to its correctness or termination properties [3]. However, he notes that the buffers required for the communication are limited by nature which makes it impossible to ensure delivery in all cases.

Reliable Ordering Ordering describes relationships among messages exchanged between two or more actors, i.e., whether messages arrive in the same order they were sent. This is usually limited to the order of arrival in mailboxes. Actors are free to process messages out-of-order or deploy mailboxes that sort incoming messages by priority. There are four orderings with increasingly strong assurances that we consider here: *non-deterministic*, *first in - first out*, *causal* and *total*. There are several opportunities to establish ordering. While some guarantees could be implemented by a suitable transport or application layer protocol, other require more complex synchronization between nodes.

First-in, first-out ordering (*FIFO*) means that messages 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.

The "happened before" relation [18, 21] 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 throughout the whole system.

A *total* order extends *causal* order and gives order to all messages in the system not only to causally related ones. 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 [4] 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.

The actor system Orleans [7] is an example of a framework that does not enforce ordering at all. 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 and instead relies on the ordering implicitly inherited from TCP. This leads to *causal* ordering in a local context and transport-dependent ordering for remote messaging.

Erlang and Akka both enforce *FIFO* ordering. Although Erlang defines this ordering as part of their basic rules of message passing [5], 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 [19]. In particular, system messages such as errors use special mailboxes and may be delivered out-of-order. Akka implements ordering on top of TCP, but utilizes additional per-connection queues to sort messages and handle errors such as TCP reconnects and full buffers.

Long et al. [20] explore reasons for ordering problems in message passing systems. The three main criteria they identify are 1) *synchronization*, i.e., either asynchronous or synchronous messaging, 2) *processing*, comparing non-deterministic against in-order delivery and processing, as well as 3) the *sharing* aspects data sharing and data isolation. For example, code that looks sequential but depends on asynchronous, unordered messages may lead to undefined behavior. They build a 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 models are built 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. Their evaluation shows that synchronous, in-order, and data isolation have the biggest effect on ordering problems for applications. In contrast, transitivity only helps in for very few cases. For framework designers, they see in-order delivery and data isolation as the most critical semantics. This analysis can help to weigh guarantees against their costs when choosing what to provide as a default.

Blessing et al. [8] propose implementing *causal* ordering by arranging participating nodes in a tree topology. While the approach further relies on *FIFO* ordering between each pair of nodes, it does not require additional meta data. This

[4] http://www.infoq.com/articles/no-reliable-messaging, accessed Aug'18

work related to the Pony actor language which aims to implement transparent distribution, i.e., hide the characteristics of distribution from the programmer.

3 The C++ Actor Framework

The C++ Actor framework (CAF) [10, 11] 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++, thus compilies to native code and has shown significant performance benefits. 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. CAF fits into 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 [9]. This results in a lightweight and scalable actor implementation that does not rely on system-level calls as 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. Recent optimization work by Torquati et al. [25] pushed CAF into the direction of low latency communication by reducing messaging latency by up to two orders of magnitude for low and moderate data rates.

The network stack in CAF consists of several components that manage network communication. The *middleman* provides the user-facing API of CAF in a distributed context. When communicating with actors on remote nodes, a local *proxy* is created for each directly known actor. *Brokers* are actors that abstract over a network interface and provide an actor interface for sending and receiving data. CAF deploys a system broker to parse and handle the application layer protocol BASP (Binary Actor System Protocol) for the management and communication between CAF nodes. Finally, a *multiplexer* uses a system-dependent multiplexing implementation to bridge the gap between socket operations and the broker interface.

Figure 1 shows the path that a message takes in CAF. The first step is a synchronous local operation (1). For messages to remote actors, the local proxy transparently forwards messages to the local system broker which serializes the message (2). Then, the Broker resolves the address of the receiver and transmits the message. After reception on the remote node a broker deserializes the message (3). Then, it

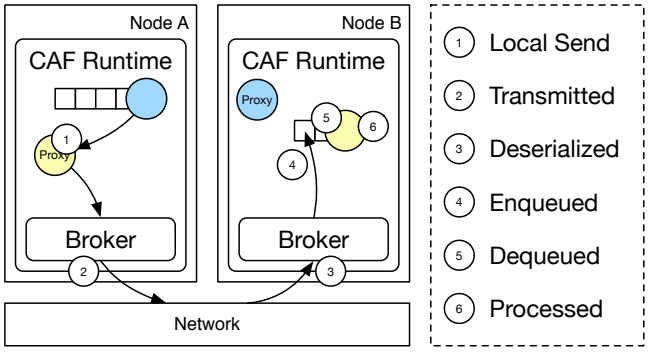

Figure 1. Message path through CAF.

is enqueued into the mailbox of the receiver (4). When the receiver is scheduled and its mailbox contains no messages that arrived previously or have a higher priority, it dequeues the message (5) and processes it (6).

4 Communication Guarantees for Actors

Message passing is the central communication primitive of the actor model for exchanging data and driving the application logic. Hence, the characteristics of the messaging layer dictate failure models and API decisions.

In this work, we focus on reliability and ordering as central aspects of any message exchange. Both concepts are well understood in packet-switching networking and implemented in transport protocols such as TCP or QUIC [17] on UDP. However, most implementations of the actor model simply rely on guarantees made by the transport protocol rather than thoroughly defining actor messaging. This is convenient for implementers of the actor model, but tightly couples fundamental system properties to deployment technologies. We leave failure detection, error propagation, reachability and security to future work. Incorporating these aspects is a natural extensions to our results presented here.

There are many choices when deciding on communication guarantees. We argue that a good design reduces complexity to a minimum for both users and implementers. Users of the system must be able to quickly form a consistent mental model without being overwhelmed by having to consider diverse edge and corner cases. Implementers likewise must be able to understand and—most crucially—debug many layers of interdependent software modules.

Reducing complexity is especially important when modeling a distributed system. Many sources of errors combined with unpredictable control flow timing pose a different challenge than designing a software stack for a single-node or even a single-thread program. Operational overhead requires careful consideration as well. Finally, incorporating a choice of desired guarantees enables developers to rely on the default implementation without adding additional hand-crafted layers on top.

This sections examines reliable delivery and ordering in regard to actor systems and discusses which guarantees impl

4.1 Reliable Message Delivery

At the lowest level, physically available memory is always limited. Messages cannot reach their destination if an actor mailbox or network buffer fails to allocate sufficient storage. However, to model delivery guarantees is still valuable for reasoning about program correctness [3].

Message delivery is reliable if each message either reaches its destination eventually or gets discarded with an error report to the sender. In other words, the system must never drop messages silently. However, the actor model is based on asynchronous message passing and does not specify errors for dropped messages. Likewise, limiting mailbox sizes is typically neither addressed nor implemented. Dedicated communication channels that signal status and demand go beyond the scope of this work but are addressed by the forthcoming streaming API in CAF.

Messages to remote actors travel through several software layers until they reach the destined actor. Figure 1 depicts this path specifically for CAF, but each implementation of the actor model will have similar steps. Hence, using CAF for examining individual steps and discussing algorithm choices translates well to other systems.

Local Sending is a synchronized operation that only fails when running out of memory. Remote actors are represented by proxies that transparently forward messages—along with meta information for reaching the remote actor—to the system broker. Mailbox state of remote actors remains opaque, as proxies act only as a message relay. Tightly synchronizing proxies with remote actors could potentially catch overloads early, but ultimately would only shift stress between nodes and impair performance by inducing very high communication overhead.

Transmitting packets to remote nodes requires peer management and serialization of messages to a portable format. The BASP broker in CAF acts as the central network hub and provides all required functionality. In particular, the broker 1) maps node IDs to sockets, 2) forwards EXIT and DOWN messages between local and remote actors, and 3) generates EXIT and DOWN messages for monitored / linked remote actors on node failures. The latter requires liveliness detection of remote nodes. Most frameworks simply rely on TCP by interpreting connection aborts as node failure. Trying to re-establish communication requires extensive buffering and sychronization when trying to maintain exactly once delivery between nodes. Alternatively, raising errors early can at least reduce the amount of buffered and potentially lost messages by giving actors immediate feedback about potentially unreachable remotes.

Deserializing at the remote node follows successful transmission. Network communication is inherently unreliable and bares additional sources of errors such as packet loss, packet duplication, or link failure. Moreover the exact failure is often hard to detect. As an example, nodes cannot distinguish between loss and delay until data arrives. Timeouts, retransmissions, and deduplication offset or solve some issues at the cost of increased communication, slow buffering, and additional complexity. Transport protocols such as TCP offer increased reliability by implementing guarantees for communication between two endpoints. Failures on the transport layer still give vague feedback to determine the liveliness of remote nodes. Initiating and managing reconnects after communication errors is not part of transport protocols. Instead, applications need to deploy necessary state and connection tracking manually. Hence, simply relying on TCP neither prevents message loss nor failures [24]. Overall, improving reliability of the network transport improves usability of the communication primitives as it relieves developers from the complexity to implement their own protocols. Deserialization fails when running out of memory. Dropping messages under temporary heavy load can become an option in the presence of an application layer protocol that handles retransmissions. Observing repeated retransmission requests from remotes also allows nodes to detect likely overloaded peers and to raise related error or status messages.

Enqueueing messages into the mailbox of the receiving actor concludes the processing steps involving brokers. Again, this operation can only fail when lacking sufficient memory. An error at this stage usually indicates an imbalance between the message arrival rate and processing capacity. Detecting and managing such issues requires some form of flow control between actors.

Dequeuing messages from the mailbox is the final step under control of the framework before handing control to user code. Estimating wait time of messages is bound to be very imprecise because it depends on processing time, fairness of the scheduling, prioritization of messages by the actor, etc. Unlike network packets, the framework could track individual messages to reproduce a global view of all messages in the system. However, considerable performance impacts due to the high synchronization overhead make global tracking undesirable in practice. The framework could still check for per-message timeouts at the point of dequeuing and drop timed-out messages. Such user-defined timeouts could force errors, but require very precise estimates by developers to add any value, in particular, trigger neither too aggressively nor too generous during ordinary program flow. In the worst case, a timeout is triggered while the response message is already traveling through the system back to the sender.

Processing messages can fail due to exceptions in user code. Such errors automatically terminate the actor and the framework propagates this failure through DOWN and EXIT messages. Estimating processing times again is very imprecise at best, unless developers have provided information for predicting runtime from message content. Actors yield

control back to the framework after completing a message, optionally producing a response message.

Discussion Three messaging steps stand out among the six that were discussed: 1) local send, 2) enqueueing into the mailbox, and 3) receiving a processing confirmation.

The first one, a "fire and forget" send, stands out because it is the simplest, most bare-bone step. Its messaging model remains asynchronous with little complexity, overhead, and state. Combined with messages that propagate liveliness of actors and nodes, complex systems can be built on top. While this approach burdens developers with error handling for basic messaging, the resulting applications are robust to a variety of failures. Most notably, this leaves the implementations with a discrepancy between local and remote message passing, thus breaking transparent distribution.

Reliable delivery to the destination mailbox extends the local guarantees to the remote messaging. The assurances of this step go beyond network transport and additionally address deserialization as well as buffering issues. The actor model does not include means to propagate these failures. Both failures categories are not easy to address generically. If deserialization fails there is no solution to fix it at runtime. Adding a specific message to propagate the error is possible although well defined message passing interfaces seem to be a better way to address the problem. When running out of memory a system has limited options to address the failure. Simply dropping messages that could not be processed for such a reason might allow an application layer protocol to retransmit messages until the receiver acknowledges receipt.

Acknowledging message processing provides the most insightful information about end-to-end communication [22]. At the same time, addressing a generalized use-case is a very complex task heavily dependent on the application logic. 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. As a result, a reasonable failure case cannot be defined across all scenarios. Propagating related information requires messages with custom handlers at the sender side since a generic reaction cannot be assigned.

From a model perspective reliable delivery that raises remote to local guarantees is valuable for modeling and makes it easier for developers to argue about their code. In practice, the step from delivery over the network to enqueuing messages into mailboxes does not provide enough benefit to merit an additional application layer protocol. Cases that merit overhead to ensure delivery are often interested in the processing results and not only the delivery, thus falling into the category of the end-to-end argument.

4.2 Reliable Message Ordering

Reading code and understanding side effects is easier when messages sent by sequential statements are delivered in the same order [20]. Relying on the same ordering for local and 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 predictable. While priority messages naturally break ordering, users expect that effect.

Non-deterministic ordering is easy to implement. Although dependent on the implementation details of local message passing, this often leaves developers with different guarantees for local and remote communication [16].

First in, first out (FIFO) ordering can be implemented for actor-to-actor or node-to-node communication. It requires sequence numbers to determine order and buffering to restore it in both cases, but with differing granularity. Implementing ordering per actor distributes the problem and avoids ordering unrelated messages between different actors. In practice, this not only introduces an additional step between the application layer protocol and actor messages, but requires state that scales with the number of actors in the system. Moving ordering to a protocol between each pair of nodes offers much better scalability as the state to track sequence numbers and buffers only scales with the number of peers. On the downside, a delayed message also impacts unrelated messages.

Causal ordering can be established in various ways. Restricting communication to synchronous message passing is often easy to implement, but heavily impacts the application behavior. The asynchronous nature of actor messages does not map well to such a restriction. Annotating each message with metadata is another option. The additional information that needs to be exchanged are time vectors with a size equal to the number of processes n [13]. Moreover, message sizes increase further to determine *causal* dependencies for transitive message passing with more than one intermediate node [2]. A third alternative is a fixed routing topology such as a ring or a tree as discussed in the context of the Pony language [8]. This overloads routing and leads to a worst cases where messages are routed from one leaf through the root to another leaf, thus introducing latency. Maintaining the topology with nodes joining, leaving, or failing is a complex task that becomes more tedious in mobile environments.

Total ordering requires a straight forward but very expensive implementation. One node in the system is chosen as a sequencer that determines the message order. Such an approach introduces a strong coupling in the system as even local messages would have to be subject to this process.

Discussion Local delivery in CAF leads to a *causal* ordering of 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.

Total order is not a desirable property for messages exchanged between actors. By definition, actors are concurrent

and isolated entities. Adding a strong coupling in the form of a central sequencer to all communications impacts scalability and performance without significant benefit. 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 default ordering.

Implementing *causal* ordering also comes at significant cost. Relying on synchronous communication introduces a strong coupling between actors and nodes. Although synchronization on a local machine is cheap, extending it to a remote context significantly impacts performance and scalability. The cost of synchronization over the network is several orders of magnitude higher and introduces undesirable delay. Developers would have a strong incentive to avoid remote communication breaking transparency on another axis. Alternatively, adding vector timestamps to messages largely increases the amount of data exchanged in the system. In addition, hosts schedule high amounts of actors and frequently spawn 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 approach comes without tradeoffs that significantly impact performance and scalability of an actor system.

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. For each pair of actors reasoning about exchanged messages is straight forward. As such it is a tradeoff between desirable properties and overhead.

5 A Composable Network Stack

Maintaining a consistent set of communication guarantees across exchangeable transport protocols requires design changes to the CAF network stack. Although support for UDP was added recently, developers who want to integrate new transport protocols are still required to adjust various components throughout the I/O library. Extending the guarantees of transport protocol requires implementation on top of a broker and is not easily reusable.

The redesign addresses these issues and leads to a composable network stack that can be extended with new transport protocols and augmented with reusable protocol layers to add to its functionality. With the goal to enable use of arbitrary transport protocols, this concept uses TCP and UDP as examples for the design. These two protocols do not cover all functionality that transport protocols can offer, but differ greatly in their included guarantees. While UDP is a barebones protocol that provides connectionless transmission of datagrams with few guarantees, TCP streams bytes with strong reliability and ordering guarantees among others. Thus, this protocol selection provides the opportunity to examine how our concept could integrate them.

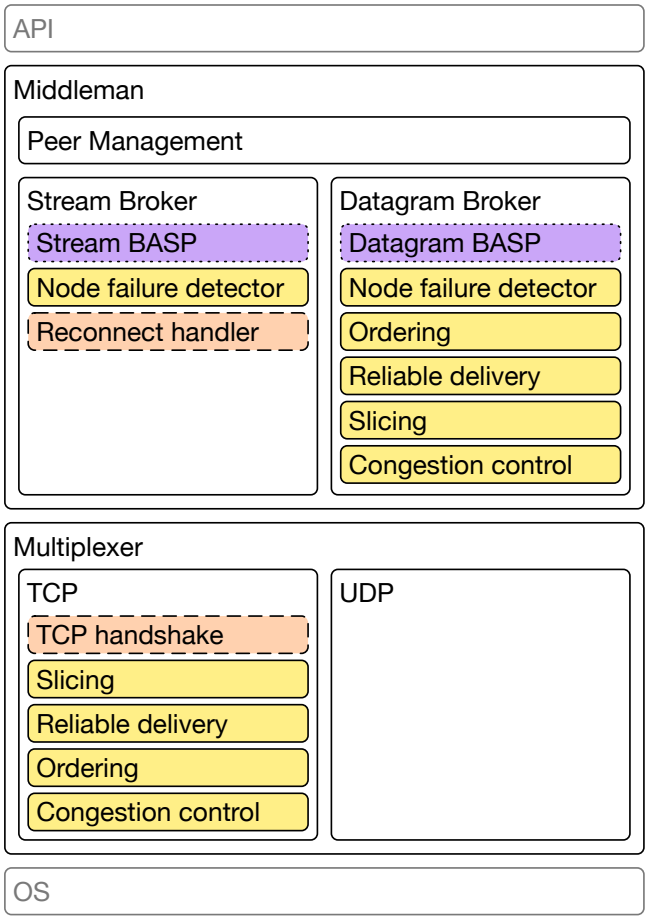

Figure 2. Composition of the CAF network stack deploying TCP and UDP.

The design of the network stack is shown in Figure 2. Yellow boxes (normal border) signify general functionality that can be provided by and for various protocols. Orange boxes (dashed border) are TCP specific and purple boxes (dotted border) are specific to CAF. Management of sockets is handled by the *multiplexer* which interfaces with the OS to provide asynchronous socket access. Located above the multiplexer in the stack are *brokers* which are managed by the *middleman*. A broker bundles a transport protocol with additional layers and wraps it in a message passing interface.

In the example case, a TCP-based stream broker adds layers to detect remote node failures and attempt reconnects in addition to a component for reading and writing BASP messages from and to a byte stream. The second broker handles datagrams characteristic for UDP. Similar to the stream broker it deploys a failure detector and a layer to translate between datagrams and BASP messages. Additionally, it is extended to slice messages into datagrams of suitable size to avoid IP fragmentation, order incoming datagrams and ensure their delivery. A reconnect handler is not needed due the connectionless nature of UDP. While some layers might be valuable in both protocols, they could benefit from

a protocol specific implementation. As an example, a TCP failure detector could monitor the connection state to detect failures.

Brokers are suitable components for this functionality. They sit in-between the actor abstraction and low-level socket API. As such, they already require protocol dependent code to translate between the incoming bytes and application data. Placing such functionality higher up would make it part of the application logic. While developers are free to do so, the default approach should cleanly separate the networking logic from the application logic. In contrast moving lower down the stack hinders access as the functionality would be colocated with low level code. Brokers are a fitting abstraction for this task.

Implementation In CAF, a broker is a component that abstracts over an endpoint for a specific transport protocol. Instead of running in the system scheduler, it is scheduled in the event loop of the multiplexer. The multiplexer executes it when an I/O event occurs on its socket or when it receives a message.

There are two types of brokers. The first one handles regular events on a socket. Similar to other actors it handles messages according to its behavior which has to include a handler for the message type it receives for new data on its sockt. It is configured by two policies: a *transport policy* and a *protocol policy*. Policies are a way to implement configurable components in C++.

A *transport policy* wraps a transport protocol by implementing functionality to read from and write to a socket and manage the related buffers.

The layers that augment guarantees or functionality are thereby define the overall protocol are bundled in a *protocol policy*. Each layer accepts the type of the next layer as a template argument and instantiates it as a member. An exception is the upmost layer which does not have another layer as a member. In addition, it dictates the message type passed to the broker for new data.

Before sending data, the protocol policy gives each layer the opportunity to write headers, set timeouts, and augment the send buffer. Similarly, upon receipt each layer can read its header, set timeouts, and sent messages. The order of layers is meaningful and is reverse for sending and receiving.

The second broker type is responsible for accepting new endpoints or multiplexing over a single socket, if desired. It creates a new broker of the first type to handle new endpoints. It is configured by an *accept policy* that determines how to react to incoming data. For TCP, an accept policy could simply accept new connections and pass the sockets to its broker, which in turn spawns brokers to handle regular communication.

6 Evaluation

Network performance is critical when building a framework that enables horizontal and vertical scalability. Using C++ for such a task further raises the expectation that the implementation performs well and its abstraction comes at little cost. Our initial evaluations focus on the cost of layers in a composable network stack.

6.1 Experimental Setup

Measurements were performed on a 2017 MacBook Pro with a 2.9 GHz Intel Core i7 and 16 GB RAM running macOS 10.14. The benchmark § 6.3 uses Mininet [14] to simulate a network link with loss. Mininet offers a VM image[5] with a configured environment. We used the image running in Virtual Box Version 5.2.18 to perform all benchmarks.

Our benchmarks can be found online on GitHub[6] and are based on the CAF branch linked in the repository. For § 6.2 we used Google Benchmark[7] in version 1.4.1 to perform the measurements.

6.2 The Costs of Layers in CAF

Passing data through the layers of a protocol policy happens on every send and receive call. Quantifying the time spent in the new broker class when sending and receiving data is valuable to evaluate the implementation in general and find performance problems. The measurements in this section do not include calls to the socket API. Instead, a mock transport policy offers buffers to read from and write to. Benchmarks that send data write their header and payload into a buffer of the policy, thus introducing a dependency between runtime and payload size. Since the mock data that is received can mostly be prepared in advance, benchmarks that receive data only have to copy data that they require to parse headers.

All benchmarks were performed for payload sizes from 128 to 8,192 bytes in increasing powers of two. All graphs show the mean real time in microseconds over ten runs as a function of the payload size and plot the standard deviation as error bars.

Sending The first benchmark examines the cost to prepare a message for sending. It compares the policies used for the TCP and UDP-related implementations. For both protocols we measure the operation cost to handle a raw protocol that does nothing but write to the send buffer and a simplified BASP protocol that prefixes data with a header consisting of a source and destination actor as well as a payload size. The UDP measurements additionally include an ordering layer that adds a sequence number.

Figure 3 shows the results for TCP on the left and UDP on the right. The time to send data rises linearly with the payload size due to the copy operations. As expected, using

[5] http://mininet.org/download/
[6] https://github.com/inetrg/agere-2018
[7] https://github.com/google/benchmark

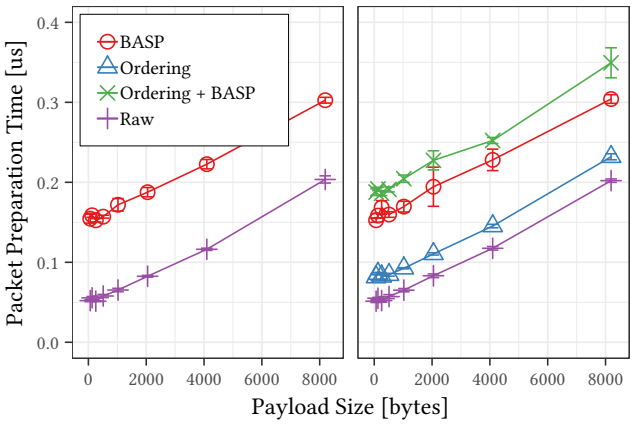

Figure 3. Cost to prepare a message for sending with different protocol layers. (left: TCP, right: UDP)

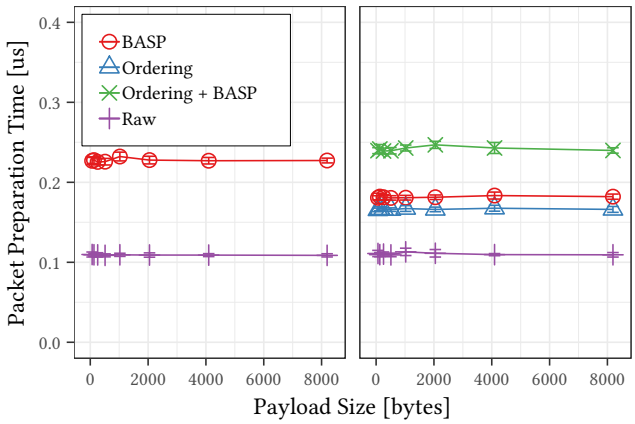

Figure 4. Cost to prepare a received packet for processing with different protocol layers. (left: TCP, right: UDP)

the raw protocol induces the least overhead in both cases with similar time requirements for both protocols.

Adding a layer introduces additional overhead depending on the layer implementation. The BASP layer comes at the same cost for both protocol. The additional time requirements stem from the serialization of its three fields: actor ids (64 bit) of the sender and receiver as well as a size parameter (32 bit).

UDP additionally includes measurements for ordering. The ordering header is smaller than the BASP header, only including a single sequence number (16 bit). The cost for adding ordering seems constant, whether it is deployed only with the raw protocol or in addition to BASP.

The error bars are small overall with the exception of a few measurement points. BASP for UDP with a payload size of 2000 bytes and 4000 bytes shows small error bars as does ordering with BASP for UDP with a payload of 8000 bytes. This could be a result of measurements on such a small time scale.

Receiving In general, message receipt promises to show a greater impact on performance. Depending on the protocol, it requires not only deserialization and parsing of the protocol headers but may include checks such as the validation of sequence numbers for ordering. In this benchmark the packet to receive is prepared in advance but adjusted during each receive call to include the expected sequence number and payload size. This means that no message is received out of order.

Figure 4 depicts the time required to prepare a single received message for processing by the broker, showing TCP on the left and UDP on the right. The measurements show constant performance across all payload sizes due to the lack of a copy operation.

Once again, the raw protocol has the least overhead as it only passes a pointer to the data through the stack. The difference in performance of the raw protocol compared to the send operation (Figure 3) is likely the overhead to activate the receiving broker to handle the message with the new data.

The BASP layer has varying costs depending on the underlying transport semantics. On top of a stream protocol (left graph) it parses the stream and reads twice to parse a complete message, a first read to get the header and deserialize the payload size and a second read to get a number of matching bytes. In contrast, BASP for datagrams (right graph) expects the message to arrive in one datagram and only requires a single read as a result.

Adding the ordering layer to the datagram broker is slightly cheaper than BASP. Note that all messages arrive in order. As a results, the layer only has to check the sequence number but never perform buffering to reorder messages. The cost of ordering is approximately constant whether it is deployed only with the raw protocol or in combination with BASP.

The error bars are negligibly small for all measurements.

Receiving UDP Sequences An ordering layer that never has to reorder is very cheap. Costs only arise once packets arrive out of order or not at all. Since ordering can be deployed without reliability, missing message are dropped eventually to avoid or the message flow just stops. There are two triggers to drop a missing message: a timeout triggers or the buffer of pending messages runs full. In both cases the runtime delivers buffered messages starting with the smallest buffered sequence number. This benchmark evaluates the cost for our ordering layer to process a sequence of ten messages in three scenarios:

1. *Ordered*: All messages arrive in the expected order.
2. *Late*: One message arrives late by one.
3. *Dropped*: One message is dropped during transport.

The maximum length of the pending message buffer is configured to five messages. Timeouts are complicated to benchmark as they rely on time and are generally long compared to execution times for the operations measured here.

Thus timeouts are not represented in the benchmark. In general, triggering a timeout can be expected to be more expensive than delivering messages due to a full buffer as it requires interaction with the clock in CAF.

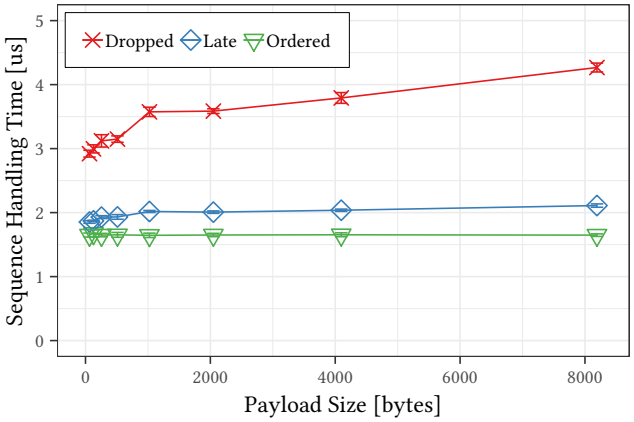

Figure 5. Cost to handle a sequence of ten packets in the presence of message loss and out-of-order delivery.

Figure 5 depicts the time required to handle the message sequence. Delivering all messages in order naturally performs best and shows a constant runtime. As soon as a message arrives late the handling time increases and no longer remains constant. Subsequent messages are buffered until the missing message is received. Since the copy operation depends on the size of the received payload we can see an increase in handling time. This behavior is more prominent when more messages need buffering. When a single message is dropped, others are buffered until the pending message buffer runs full. This behavior is hard to avoid as the bytes have to be copied from the receive buffer for later delivery. The error bars are negligible for all measurements.

6.3 Network Performance

Having implemented a composable network stack for CAF, we took the opportunity to implement a reliability layer for UDP. A virtual network built with Mininet [14] allows testing its behavior over links with configurable loss and delay. In contrast to the previous benchmarks, these measurements now include network operations.

Two brokers bounce a message back and forth 4000 times over a lossy link until each broker sent and received the message 2000 times. The Mininet topology for the benchmark consists of two hosts connected directly via a link with no delay. Our retransmit timeout for UDP is configured to be 40 ms and the minimum retransmit timeout for TCP on the routes is configured to the same value. The measurements are performed for different transport and layer combinations: TCP, reliable UDP, and reliable, ordered UDP.

Figure 6 displays the total runtime as a function of the configured packet loss percentage with error bars for the 5 and

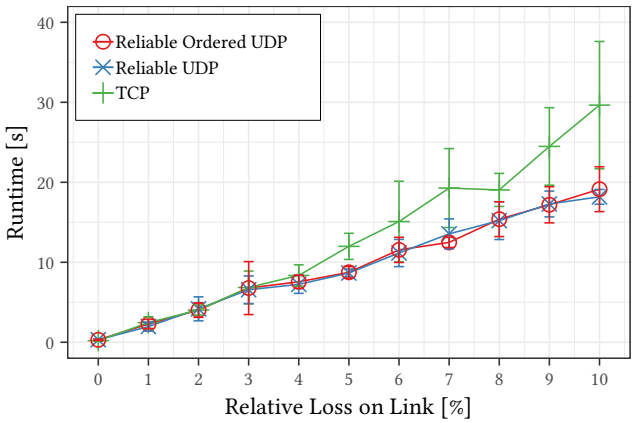

Figure 6. Two actors sequentially exchange messages over a lossy link without delay.

95 percentile. The linear increase in runtime for both UDP implementations is expected. It shows that our reliability layer performs retransmits and the program works despite the loss. Since only one message is sent at a time, every lost packet adds the retransmit timeout to the runtime. Adding the ordering layer does not impact performance as messages should not arrive out of order.

Below 3% loss TCP shows similar performance to our simple reliability layer. Thereafter, TCP is increasingly slower than UDP. A key difference here is that TCP adjusts its congestion windows and retransmit timeouts continuously in reaction to individual losses of the specific run. This is also reflected by the large error bars for TCP.

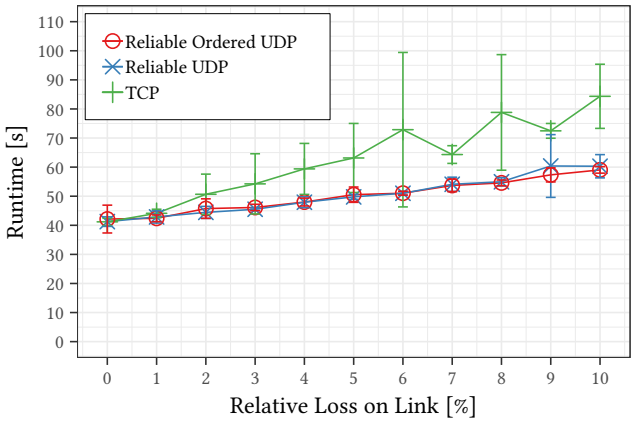

Figure 7. Two actors sequentially exchange messages over a lossy link with 10ms delay.

We repeated the benchmark in the same setup with a link delay of 10ms. Figure 7 shows the results. Note that the y-axis has a different scale. The results for UDP look similar with an offset of about 40 s. This matches the expected increase, a total of 4000 messages with 10 ms for each transmission. In contrast, the delay impacts the performance of TCP to

a greater extent. Here, the mean runtime increase is larger than for UDP.

Both protocols show larger error bars. While this can be seen for UDP especially for 0% and 9% loss, TCP shows much more variation overall and has largely increased error bars. Once again, the individual loss pattern in each run has a greater impact on TCP due to its adaptability. TCP interprets loss as network overload and re-adjusts its congestion control algorithm.

The benchmarks validate that a working retransmit protocol can be implemented as a layer in our network stack. In addition to optimizing the reliability layer for a more general use including adaptive retransmit timeouts, we want to ship a slicing layer to make it easy for users to configure the network layer for their needs.

7 Conclusion and Outlook

The characteristics of actor communication lack a common design and often change with context and implementation. Most notably guarantees often change when moving from local to remote contexts.

This work examined reliable delivery and ordering in the context of actor communication and found three notable delivery guarantees. First, a "fire and forget" approach that bares little overhead. It allows developers to build more complex systems on top but requires explicit error handling as part of the application. Second, guaranteed delivery to the mailbox of the receiving actor. This aligns the guarantees between local and remote contexts thus increasing the transparency of distribution. Third and last, guaranteed processing feedback bares great value when considering end-to-end communication. However, it induces overhead and might not be required in all cases.

With regard to ordering, the discrepancy between local and remote contexts weakens guarantees from *causal* ordering to *FIFO* or none. While algorithms exist to establish a *causal* order in distributed systems, these come at significant cost. Ensuring *FIFO* ordering already provides valuable information, helps developers to reason about their code, and comes at comparably little cost.

Many implementations inherit their guarantees for remote messaging from TCP. This is problematic as transport protocols offer more than guarantees and can adjust applications to specific environments. To enhance transport bindings in CAF we implemented a composable network stack that allows bundling a transport protocol with additional layers to add new functionality. An evaluation shows that our layer design introduces minimal overhead. Additionally a reliability layer was implemented and tested with a varying degree of packet loss to showcase a more complex layer.

Examining existing actor systems and laying out the implementation space for reliable delivery and ordering is a first step towards a more detailed discussion on the message passing guarantees for actors. While our implementation shows that a lightweight implementation is possible, generalization is required to make our results translatable to a wide range of frameworks.

There are several directions for future work. As a first step, the system broker and SSL module should be ported to the new design and thoroughly benchmarked against their previous implementations. Next, we want to examine the possibility to integrate the streaming capabilities of CAF into the new brokers. Streaming adds a backchannel to actor communication to avoid overburdening actors and with this addition could take the network behavior into account. Splitting the monolithic system broker into smaller light-weight brokers is a first step towards a multi-threaded network backend. Finally, there are aspects that were disregarded in this work and are left for future work such as reachability and security.

A Note on Reproducibility

We explicitly support reproducible research [1, 23]. Our experiments have been conducted in a transparent standard environment. The source code of our implementations (including scripts to setup the experiments, CAF measurement apps etc.) are available on GitHub at https://github.com/inetrg/agere-2018.

Acknowledgments

We are grateful for many lively discussions and the inspiring environment of the INET team in Hamburg. In particular, we want to thank Jakob Otto for his helping hands in experimentation.

This work was supported in parts by the German Federal Ministry of Education and Research within the projects Scalecast and X-Check.

References

[1] ACM. Jan., 2017. Result and Artifact Review and Badging. http://acm.org/publications/policies/artifact-review-badging.

[2] F. Adelstein and M. Singhal. 1995. Real-time Causal Message Ordering in Multimedia Systems. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS '95)*. 36–43.

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

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

[5] Joe Armstrong. 2003. *Making Reliable Distributed Systems in the Presence of Software Errors.* Ph.D. Dissertation. Department of Microelectronics and Information Technology, KTH, Sweden.

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

[7] Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. [n. d.]. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Technical Report MSR-TR-2014-41. Microsoft.

[8] Sebastian Blessing, Sylvan Clebsch, and Sophia Drossopoulou. 2017. Tree Topologies for Causal Message Delivery. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2017)*. ACM, New York, NY, USA, 1–10.

[9] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.

[10] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. 2014. 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!* ACM, New York, NY, USA, 15–28.

[11] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. 2016. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures* 45 (April 2016), 105–131.

[12] Dominik Charousset, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. 2013. 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!* ACM, New York, NY, USA, 87–96.

[13] Bernadette Charron-Bost. 1991. Concerning the Size of Logical Clocks in Distributed Systems. *Inf. Process. Lett.* 39, 1 (July 1991), 11–16.

[14] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. 2012. Reproducible Network Experiments Using Container-based Emulation. In *Proc. of CoNEXT '12*. ACM, New York, NY, USA, 253–264.

[15] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. of the 3rd IJCAI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[16] Raphael Hiesgen, Dominik Charousset, and Thomas C. Schmidt. 2016. Reconsidering Reliability in Distributed Actor Systems. In *Proc. of the 7th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '16), Poster Session*. ACM, New York, NY, USA, 31–32.

[17] Jana Iyengar and Martin Thomson. 2018. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft – work in progress 14. IETF.

[18] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.

[19] Lightbend Inc. 2018. *Akka Documentation*. https://doc.akka.io/docs/akka/current/, accessed 23-08-2018.

[20] Yuheng Long, Mehdi Bagherzadeh, Eric Lin, Ganesha Upadhyaya, and Hridesh Rajan. 2016. On Ordering Problems in Message Passing Software. In *Modularity'16: 15th International Conference on Modularity (Modularity'16)*.

[21] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. North-Holland, 215–226.

[22] Jerome H. Saltzer, David P. Reed, and David D. Clark. 1984. End-to-End Arguments in System Design. *ACM Trans. Comput. Syst.* 2, 4 (Nov 1984), 277–288.

[23] Quirin Scheitle, Matthias Wählisch, Oliver Gasser, Thomas C. Schmidt, and Georg Carle. 2017. Towards an Ecosystem for Reproducible Research in Computer Networking. In *Proc. of ACM SIGCOMM Reproducibility Workshop*. ACM, New York, NY, USA, 5–8.

[24] Hans Svensson and Lars-Åke Fredlund. 2007. Programming Distributed Erlang Applications: Pitfalls and Recipes. In *Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop (ERLANG '07)*. ACM, New York, NY, USA, 37–42.

[25] Massimo Torquati, Tullio Menga, Tiziano De Matteis, Daniele De Sensi, and Gabriele Mencagli. 2018. Reducing Message Latency and CPU Utilization in the CAF Actor Framework. In *26th Euromicro Int. Conf. on Parallel, Distributed and Network-based Processing, PDP 2018*. IEEE Computer Society, Washington, DC, USA, 145–153.