Reusable TLA+ Communication Primitives for Modeling and Verifying Distributed Systems

Diogo Canut Freitas Peixoto[1], **Odorico Machado Mendizabal**[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) Florianópolis – SC – Brasil

diogocanut1@gmail.com, odorico.mendizabal@ufsc.br

Abstract. *Challenges associated with developing distributed systems go beyond understanding business requirements, specific protocols or technologies. Modular design, inherent concurrency, and the occurrence of faults are some of the main difficulties when designing a correct and reliable distributed system. Traditional testing and debugging methods are often insufficient for distributed systems due to the complexity of reproducing rare but critical failure scenarios. Alternatively, verification techniques, such as model checking, can address these challenges by enabling the formal verification of system properties. By representing a system model and expressing temporal logic formulas, designers can identify potential safety and liveness violations. This paper introduces a modular and reusable TLA+ library for modeling communication primitives over perfect, fair-loss, and stubborn links, allowing system designers to describe and verify their solutions using these primitives as the foundation of their communication subsystem. In addition, it enables fault injection, facilitating the analysis of system behavior under unreliable communication conditions, including message drop, duplication, and out-of-order delivery. To illustrate its utility, we implemented a simple protocol and demonstrated how the assumptions and guarantees provided by the underlying communication affect correctness, even in fault-prone scenarios.*

1. Introduction

Distributed systems are fundamental to modern computing, powering cloud services, large-scale data processing, and global communication networks. However, designing correct and reliable distributed systems is notoriously difficult due to inherent challenges such as network unreliability, consistency enforcement, distributed coordination, fault tolerance, and latency constraints. These challenges are exacerbated by the system's intrinsic non-determinism, the presence of race conditions, and the exponential growth of reachable states as the system scales. Subtle design flaws may remain undetected until they cause severe failures in production, making rigorous verification techniques essential.

Traditional testing and debugging methods are often insufficient for distributed systems due to the complexity of reproducing rare but critical failure scenarios. Saltzer *et al.* [Saltzer et al. 1984] highlight the importance of reasoning about correctness at the right abstraction level rather than relying solely on low-level mechanisms. Similarly, Schneider [Schneider 1993] argues that abstraction and formal modeling are crucial for understanding and verifying complex systems. These perspectives emphasize the necessity of formal verification techniques, such as model checking, which systematically explore all possible execution paths to detect correctness violations.

Model checking enables rigorous validation of key system properties, such as safety (ensuring that *nothing bad happens*) and liveness (ensuring that *something good eventually happens*). Unlike traditional testing, which samples a subset of executions, model checking provides exhaustive analysis within a formally defined specification [Baier and Katoen 2008]. One such formal method is TLA+ (Temporal Logic of Actions) [Lamport 1994], a specification language designed for reasoning about concurrent and distributed systems. By formally specifying system behavior, TLA+ allows designers to identify safety and liveness violations early in the design process, reducing costly late-stage errors.

In this work, we introduce a modular and reusable TLA+ library for modeling communication primitives in distributed systems. This library provides system designers with a structured approach to specifying and verifying distributed protocols while abstracting away low-level communication details. Our library supports point-to-point communication over perfect, fair-loss, and stubborn links.

It also enables fault injection, allowing researchers and practitioners to observe system behavior under unreliable communication conditions, including message drop, duplication, and out-of-order delivery. Finally, we demonstrate the usage of the proposed library with a simple use case, where the echo protocol is modeled. Changing underlying communication guarantees makes it easy to observe how and why protocol properties are violated.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 introduces common system model assumptions and definitions in distributed systems. Section 4 presents the communication models considered in this work. Section 5 details the implementation of our TLA+ library, and Section 6 illustrates its application through a case study. Finally, Section 7 concludes the paper.

2. Related Work

Model checking is a well-established technique in protocol and system verification, with numerous contributions advancing the state of the art in distributed systems verification. Of particular relevance to this work, TLA+ has demonstrated versatility in verifying a wide range of distributed protocols. For instance, it has been used to specify and verify the Pastry protocol, which provides a distributed hash table over a peer-to-peer network [Lu et al. 2011], as well as Tendermint, an open-source Byzantine Fault Tolerant consensus engine for blockchain systems [Braithwaite et al. 2020]. Additionally, TLA+ has been employed to verify Zookeeper's atomic broadcast protocol [Yin et al. 2020], ensure security properties of smart contracts [Kolb et al. 2020], and model the Lightning Network protocol [Grundmann and Hartenstein 2023].

These examples illustrate the broad applicability of TLA+ in distributed system verification. However, they are typically domain-specific and require a full specification of each target application, including its semantics, communication, and fault behavior. Our approach differs by providing a modular and reusable library for point-to-point communication, where fault behaviors and different guarantees are predefined, eliminating the need for system designers to model them explicitly.

Automatic fault injection in model checking has been explored in prior work. For example, [Dotti et al. 2005] employs Object-Based Graph Grammars (OBGG) to model

fault-tolerant distributed systems, introducing crash and omission faults via automatic specification translation. In contrast, our approach does not require specification transformation; instead, we provide an API for message transmission, allowing designers to select communication assumptions directly.

Similarly, [House and Tang 2018] presents a reusable TLA+ module for modeling asynchronous message-passing systems, assuming reliable channels and using graph-based representations of communication structures. While also benefiting from modularity, their approach does not generalize to unreliable communication links. Our work was further inspired by [dmilstein 2019], which provides reusable TLA+ modules with guarantees such as out-of-order delivery and message duplication. However, unlike our approach, it does not account for message loss or provide a link abstraction for different communication properties.

3. System models and assumptions in distributed systems

In point-to-point communication, a distributed system is represented as a graph, where nodes correspond to system processes and edges represent communication links between them. Each link connects two nodes through their network interfaces, enabling message exchange.

Distributed systems are classified based on timing assumptions into synchronous, asynchronous, and hybrid models. In a synchronous system, upper bounds exist on message transmission delays, process execution times, and clock drift, allowing for precise coordination. In contrast, an asynchronous system imposes no timing constraints—messages may experience arbitrary delays, processes execute at unpredictable speeds, and there is no global clock, making failure detection more challenging. Time-synchronous and partially synchronous models lie between these extremes, assuming partially known timing bounds, such as loosely synchronized clocks, probabilistic message delays, or alternating synchronous and asynchronous behavior. These hybrid models enhance coordination while accommodating some level of unpredictability. A comprehensive survey on system models and assumptions is presented in [Gärtner 1999].

Failures in distributed systems stem from hardware faults (e.g., network or node failures) and software faults (e.g., design or implementation errors). To abstract system-specific behaviors, failures are commonly categorized into the following classes:

- **Fail-Stop** The affected component halts execution, and its failure can be detected by other processes;
- **Crash** The component stops or loses its internal state, but unlike fail-stop failures, its failure is not detectable by others;
- **Send Omission** A process fails to send a subset of messages. This model may exhibit crash-like behavior (e.g., if all sent messages are dropped);
- **Receive Omission** A process fails by receiving only a subset of expected messages. This model also may exhibit crash-like behavior (e.g., if all received messages are dropped);
- **General Omission** In the general omission model, both send omission and receive omission are present;
- **Byzantine or Arbitrary** Nodes exhibit unrestricted behavior, potentially sending conflicting or malicious messages.

Our approach does not model *fail-stop*, or *Byzantine failures*. *Fail-stop* is excluded as it is overly restrictive, with *crash* failures being more commonly assumed in distributed systems literature. *Byzantine failures* depend on application semantics and are thus beyond the scope of our reusable library.

4. Communication primitives

In distributed systems, processes interact through message passing to share data, coordinate actions, and maintain consistency among replicas. This interaction relies on communication primitives, which serve as low-level abstractions providing fundamental messaging capabilities. Practical implementations of these primitives vary in their reliability guarantees, including message ordering, fault tolerance, and delivery integrity.

For every pair of communicating processes, there exists a communication channel that facilitates message exchange. These channels are abstracted to model the properties and behavior of the underlying communication infrastructure. In real-world systems, messages can be lost due to network failures, delays, or transient issues. However, the probability of a message successfully reaching its destination is nonzero, as long as the network is not experiencing severe or permanent failures.

Point-to-point communication is commonly referred to as a link, with its properties defined using two event abstractions: *Send* and *Receive*. The *Send* event represents a source process initiating message transmission over a link to a destination process. The link is then responsible for delivering the message according to its specified guarantees. In this paper, we use *Receive* to denote the action of passing a received message to the destination process, whereas other works may refer to this as the Deliver event.

4.1. Perfect Link

The Perfect link abstraction, also referred to as a *Reliable link*, builds upon the basic properties of no duplication and no creation by adding the property of reliable delivery. This means that every message sent by a correct sender to a correct receiver is guaranteed to be delivered exactly once, provided both processes are correct. This abstraction ensures that no messages are lost or invented, and each message is delivered in the same form it was sent. In [Cachin et al. 2011], the authors define perfect link properties as follows:

Property 1 (Reliable delivery) *If a correct process p sends a message m to a correct process q, then q eventually delivers m.*

Property 2 (No duplication) *No message is delivered by a process more than once.*

Property 3 (No creation) *If a process q delivers a message m with sender p, then m was previously sent to q by process p.*

4.2. Fair-loss link

The fair-loss link assumes that messages might be lost due to network issues. However, the probability of a message being delivered is nonzero if it is sent infinitely often. This means that if the sender continues to retransmit a message, there is a guarantee that the message will eventually be delivered. Additionally, it is assumed that no message is created or corrupted by the network, and the network does not transmit more messages than those originally sent by the sending process. In [Cachin et al. 2011], the authors define the properties as follows:

Property 1 (Fair-loss) *If a correct process p infinitely often sends a message m to a correct process q, then q delivers m an infinite number of times.*

Property 2 (Finite duplication) *If a correct process p sends a message m a finite number of times to a process q , then m cannot be delivered an infinite number of times by q.*

Property 3 (No creation) *If a process q delivers a message m with sender s, then m was previously sent by process s.*

4.3. Stubborn link

The Stubborn link is an extension of the Fair-loss link that hides the retransmission mechanisms used by the sender. While Fair-loss links allow messages to be lost, Stubborn links ensure that the sender continues to retransmit messages indefinitely, increasing the likelihood of eventual delivery. This extension retains the no-creation property, ensuring that no new messages are invented by the link. Additionally, Stubborn links differ by allowing the same message to be delivered to the receiver multiple times, unlike Fair-loss links which do not implement retransmissions [Cachin et al. 2011].

Property 1 (Stubborn delivery) *If a correct process p sends a message m once to a correct process q, then q delivers m an infinite number of times.*

Property 2 (No creation) *If a process q delivers a message m with sender s, then m was previously sent by process s.*

5. The TLA+ Communication Module

This section presents our TLA+ module implementation, which provides a reusable implementation of common abstractions used for channels over various communication links.

TLA+ (Temporal Logic of Actions) is a formal specification language designed for modeling and verifying concurrent and distributed systems [Lamport 1994]. It combines first-order logic, set theory, and temporal logic to describe system behaviors, enabling rigorous reasoning about correctness. The language supports state-based modeling, where system states and transitions are explicitly defined, and modular specification, allowing for reusable components. To verify system properties, TLA+ employs Linear Temporal Logic (LTL), enabling the formal expression of safety and liveness properties. Key temporal operators include $\Box$ (always), $\Diamond$ (eventually), and $\rightarrow$ (leads to), which facilitate reasoning about system evolution over time.

The TLC model checker is a verification tool that systematically explores all possible system executions to detect violations of specified properties. It supports exhaustive and bounded model checking, simulation-based verification, and state-space reduction techniques to handle large models. TLC can identify issues such as deadlocks, safety violations, and invariant failures, providing counterexamples that illustrate erroneous executions.

In TLA+, point-to-point communication between processes is often modeled using channels. However, accurately modeling communication links can be challenging due

to the variety of reliability guarantees required, such as handling message loss, duplication, or eventual delivery. To address these challenges, this work introduces a communication module that abstracts the behavior of different types of communication links, offering reusable building blocks for specifying and verifying distributed systems in TLA+.

5.1. Modularity and reuse

TLA+ allows one to create reusable pieces of code that can be composed to model complex systems. Through its modular design capabilities, TLA+ supports the definition of parameterized modules and operators, enabling encapsulation of common patterns and abstractions. There is also the possibility to define constants during module extension, which gives users the flexibility to adapt the module to their specification's scale.

5.2. Faulty behavior

Faulty behavior in distributed systems arises due to the inherent challenges of asynchronous communication lacking a global clock. As discussed in [Gärtner 1999], fault-tolerant systems are designed to handle various types of faults, categorized broadly as crash faults, omission faults, and Byzantine faults. Handling each type introduces unique challenges and requires distinct strategies to ensure reliability and correctness.

In our specifications, faulty behavior is inspired by the strategy outlined in [Gärtner 1999, Dotti et al. 2005]. Omission faults are modeled by allowing messages to be dropped in unreliable communication links, as demonstrated in our Fair-Loss Link module. Additionally, we conducted our case studies under two scenarios: with process fairness and without fairness. In the absence of fairness, crash faults were simulated by allowing a process to halt execution at any point during the execution. This halt behavior is naturally captured by TLC through the inclusion of stuttering steps, which model the possibility of a process ceasing to make progress without violating the temporal logic semantics.

5.3. Application Programming Interface and Implementation

The initial focus of this work was on the implementation of three types of communication links: the Perfect Link, the Fair-Loss Link, and the Stubborn Link, each with distinct guarantees and behaviors. The Perfect Link ensures reliable communication by guaranteeing that messages are neither lost nor duplicated and are always received in the order they are sent. In contrast, the Fair-Loss Link simulates unreliable communication, allowing for message loss and providing no guarantees regarding message ordering or duplication. Finally, the Stubborn Link guarantees eventual delivery of messages without loss but permits message duplication and does not enforce any ordering.

Figure 1 depicts an high-level overview of the abstracted API interface and the underlying communication modules. Users can describe their distributed algorithms or protocols using our TLA+ communication module, where primitives *Send* and *Receive* are available through operators for each link. Thus, these operators encapsulate the underlying behavior of the links while presenting a uniform interface to the user. This abstraction allows system designers to model communication patterns without being burdened by the implementation details of each link. When modeling, the only differences lie in the specific parameters the *Send* and *Receive* operators require for each link type. Therefore,

users can easily switch between different underlying communication links to evaluate how their distributed protocol behaves under various reliability assumptions, without needing to modify the core logic of their specification.

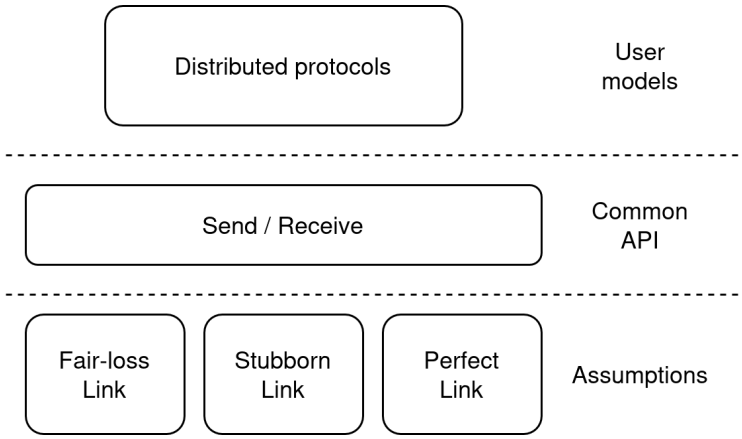

Figure 1. Module Interface

5.4. Perfect Link Module

For modeling a channel in TLA+ we created a mapping that associates each process in the set $processes$ to an initially empty sequence. We used the set to represent the inbox of the channel of each process. The definition is as follows:

$$PerfectLink(processes) \triangleq [p \in processes \mapsto \langle\rangle]$$

For those who are not familiar with TLA+, this is the definition of an operator called $PerfectLink$, which takes a parameter $processes$ (typically a set of process identifiers). The right-hand side defines a function (or mapping) that associates each process p in the set $processes$ with an empty sequence. In TLA+, sequences are used to represent ordered collections of elements, and the empty sequence $\langle\ \rangle$ denotes that no messages have yet been received.

To send messages, we defined a $Send$ operator that models the act of delivering a message to a specific receiver by appending it to the receiver's inbox. In TLA+, complex operators are often supported by auxiliary definitions, and in our case, $Send$ relies on two local helper functions: $WrapMessage$ and $AppendMessage$.

The $WrapMessage$ operator constructs a structured record that encapsulates the sender, receiver, and the actual message. This wrapping provides contextual information with each message, which can be useful when handling more advanced features such as tracing, deduplication, or error handling. The $AppendMessage$ operator takes the current link state, retrieves the inbox of the receiver, and appends the wrapped message to it using TLA+'s Append function. These two helper functions are declared as $LOCAL$, meaning they are internal to the module and not visible when the module is extended or instantiated elsewhere. This allows us to encapsulate implementation details while exposing only the $Send$ operator as the public interface.

$$
\begin{aligned}
&\text{LOCAL } \mathit{WrapMessage}(\mathit{sender}, \mathit{receiver}, \mathit{msg}) \triangleq \\
&\quad [\mathit{sender} \mapsto \mathit{sender}, \mathit{receiver} \mapsto \mathit{receiver}, \mathit{message} \mapsto \mathit{msg}] \\
&\text{LOCAL } \mathit{AppendMessage}(\mathit{link}, \mathit{sender}, \mathit{receiver}, \mathit{msg}) \triangleq \\
&\quad \mathit{Append}(\mathit{link}[\mathit{receiver}], \mathit{WrapMessage}(\mathit{sender}, \mathit{receiver}, \mathit{msg})) \\
&\mathit{Send}(\mathit{link}, \mathit{sender}, \mathit{receiver}, \mathit{msg}) \triangleq \\
&\quad [\mathit{link} \text{ EXCEPT } ![\mathit{receiver}] = \\
&\qquad \mathit{AppendMessage}(\mathit{link}, \mathit{sender}, \mathit{receiver}, \mathit{msg})]
\end{aligned}
$$

To deliver a message, the $\mathit{HasMessage}$ operator checks whether a message is available in the inbox, while the $\mathit{Message}$ operator retrieves the first message in the inbox. The ***Receive*** operator then removes the message from the inbox, marking it as successfully delivered. Additionally, since we have added extra information to the message structure, we use the $\mathit{UnwrapMessage}$ operator to extract the original message content.

$$
\begin{aligned}
&\mathit{UnwrapMessage}(\mathit{wrappedMessage}) \triangleq \mathit{wrappedMessage}.\mathit{message} \\
&\mathit{HasMessage}(\mathit{link}, \mathit{process}) \triangleq \mathit{link}[\mathit{process}] \neq \langle\rangle \\
&\mathit{Message}(\mathit{link}, \mathit{process}) \triangleq \mathit{Head}(\mathit{link}[\mathit{process}]) \\
&\mathit{Receive}(\mathit{link}, \mathit{process}) \triangleq \\
&\quad [\mathit{link} \text{ EXCEPT } ![\mathit{process}] = \mathit{Tail}(\mathit{link}[\mathit{process}])]
\end{aligned}
$$

5.5. Fair-loss Link Module

There are several differences between the implementations of the Perfect Link, Fair-Loss Link, and Stubborn Link. Instead of using a sequence to define the inboxes, both the Fair-Loss and Stubborn Link implementations use a set. This allows for the simulation of out-of-order delivery; however, since a set does not allow duplicates, it is necessary to ensure that messages are uniquely identifiable. To achieve this, each message is wrapped with an incremental message ID.

In the Fair-Loss Link, it is necessary to track the total number of messages dropped. For this purpose, a variable named ***totalDrops*** is added to the link and is updated during each $\mathit{DropMessage}$ operation. Additionally, to keep track of the next incremental integer ID, a variable named ***nextMessageId*** is included. This variable is incremented with each Send operation to ensure that every message has a unique identifier.

$$
\begin{aligned}
&\text{LOCAL } \mathit{InitLink}(\mathit{linksPerProcess}) \triangleq \\
&\quad [\mathit{links} \mapsto \mathit{linksPerProcess}, \mathit{nextMessageId} \mapsto 0, \mathit{totalDrops} \mapsto 0] \\
&\mathit{FairLossLink}(\mathit{processes}) \triangleq \mathit{InitLink}([p \in \mathit{processes} \mapsto \{\}])
\end{aligned}
$$

To mitigate state space explosion, we define a constant called ***MaxDrops***, which serves as the upper limit for the number of messages that can be dropped. During each invocation of the Send operator, we check whether ***totalDrops*** has reached or exceeded ***MaxDrops***. If it has, further message drops are halted to ensure the limit is not exceeded.

$$
\text{LOCAL } \mathit{ShouldDrop}(\mathit{link}) \triangleq \mathit{link}.\mathit{totalDrops} < \mathit{MaxDrops}
$$

We delegate the handling of message-dropping non-determinism to the end user, allowing them to manage it outside the module. This approach enables the designer to decide when a message should be dropped, based on their own fairness specifications. The Send operation is defined as follows:

$$
\begin{aligned}
&\mathit{Send}(\mathit{link}, \mathit{sender}, \mathit{receiver}, \mathit{msg}, \mathit{nonDeterministicShouldDrop}) \triangleq \\
&\quad \text{IF } \mathit{nonDeterministicShouldDrop} \land \mathit{ShouldDrop}(\mathit{link}) \\
&\quad \text{THEN } \mathit{DropMessage}(\mathit{link}) \\
&\quad \text{ELSE } \mathit{ReliableSend}(\mathit{link}, \mathit{sender}, \mathit{receiver}, \mathit{msg})
\end{aligned}
$$

Both the implementations of $DropMessage$ and $ReliableSend$ return the link mapping structure. The difference is that $DropMessage$ does not add the new message to the set and increments the ***totalDrops*** value, whereas $ReliableSend$ updates the receiver's inbox by adding the new message and increments the ***nextMessageId***.

$$
\begin{array}{l}
\text{LOCAL } DropMessage(link) \triangleq [\\
\quad links \mapsto link.links, \\
\quad nextMessageId \mapsto link.nextMessageId, \\
\quad totalDrops \mapsto link.totalDrops + 1] \\
\text{LOCAL } ReliableSend(link,\ sender,\ receiver,\ msg) \triangleq [\\
\quad links \mapsto [link.links \text{ EXCEPT } ![receiver] = \\
\qquad AppendMessage(link,\ sender,\ receiver,\ msg)], \\
\quad nextMessageId \mapsto link.nextMessageId + 1, \\
\quad totalDrops \mapsto link.totalDrops]
\end{array}
$$

Both the Stubborn Link and Fair-Loss Link share the same interface for delivering messages. The first operator, $HasMessages$, is used to check whether the process inbox set is empty or contains any messages. Next, the $Messages$ operator can be used to retrieve the set of messages currently in the process inbox. Finally, the $Receive$ operator is used to deliver a single message and remove it from the inbox. It is the user's responsibility to ensure that every message read from the inbox is eventually delivered.

$$
\begin{array}{l}
Messages(link,\ process) \triangleq link.links[process] \\
Receive(link,\ process,\ wrappedMessage) \triangleq [\\
\quad links \mapsto [link.links \text{ EXCEPT } ![process] = \\
\qquad \{m \in link.links[process] : m \neq wrappedMessage\}], \\
\quad nextMessageId \mapsto link.nextMessageId, \\
\quad totalDrops \mapsto link.totalDrops]
\end{array}
$$

5.6. Stubborn Link Module

The main difference between the Stubborn Link and the Perfect Link is that the Stubborn Link is an implementation of the retransmit-forever algorithm over a Fair-Loss Link, which can lead to potential duplication of messages but ensures no message loss. In the Stubborn Link specification, we define a constant called ***MaxCopies***, which specifies the maximum number of times a single message will be duplicated. We define a maximum number of copies to avoid state space explosion.

Additionally, a new field is added to each message to represent the copy number. The duplication behavior is defined in the $Send$ operator by simulating the retransmission of the message up to the ***MaxCopies*** limit. The out-of-order behavior is preserved by employing the same strategy as in the Fair-Loss Link, where a set is used to model the link for each process. To support this functionality, we define an operator called $DuplicableAppendMessage$.

$$
\begin{array}{l}
\text{LOCAL } DuplicableAppendMessage(link,\ sender,\ receiver,\ msg) \triangleq \\
\quad (link.links[receiver] \cup \\
\qquad \{WrapMessage(sender,\ receiver,\ msg,\ link.nextMessageId,\ copy) \\
\qquad\quad : copy \in 1\ ..\ MaxCopies\})
\end{array}
$$

We then define the operator $DuplicableSend$ using $DuplicableAppendMessage$, such that it transforms the underlying link structure by appending all copies of the message and increments the ***nextMessageId***.

$$\text{LOCAL } DuplicableSend(link,\ sender,\ receiver,\ msg) \triangleq$$
$$[links \mapsto [link.links \text{ EXCEPT } ![receiver] = DuplicableAppendMessage(link,\ sender,\ receiver,\ msg)],$$
$$nextMessageId \mapsto link.nextMessageId + 1]$$

Finally, we define the $Send$ interface as

$$Send(link,\ sender,\ receiver,\ msg) \triangleq$$
$$DuplicableSend(link,\ sender,\ receiver,\ msg)$$

6. Case studies: Verification of distributed protocols

To verify the implementation of the communication module, we utilized the PlusCal language [Lamport 2009], which resembles a procedural programming language and can be translated into TLA+ for formal verification using the TLC model checker. We modeled a version of the Echo Protocol, which operates as follows:

Process A sends a message to *Process B* and waits synchronously for a reply from B. Upon receiving the message, *Process B* forwards it back to *Process A*. The messages consist of a predefined sequence of integers, with the number -1 used to signify the end of the connection, acting as a FIN (Finish) signal. When *Process B* receives a FIN signal, it forwards the signal back to *Process A* and terminates its execution. Similarly, when *Process A* receives the FIN message in response, it also terminates.

Additionally, we defined three essential properties that the Echo Protocol implementation must satisfy. These properties ensure the protocol's correctness under varying conditions and guarantee its expected behavior. The properties are as follows:

Property 1 (Delivery Guarantee) *If process A sends a message m, then process A eventually delivers m.*

Property 2 (Termination) *Eventually both process A and B will receive the FIN signal and terminate.*

Property 3 (Causal Consistency) *If a process A delivers a message m, it means that process A sent m.*

We also utilized reusable design patterns to express Linear Temporal Logic (LTL) properties, as outlined in [Salamah et al. 2005]. For our first property, we employed the Universality Pattern, which ensures that a specified condition holds true across all system states. The adoption of design patterns in LTL promotes consistency in specifying behaviors, and facilitates their application across various contexts. Below, we present the formal specifications of the desired echo properties using LTL.

Property 1 (Delivery Guarantee)

$$\Box(\textit{messageToSend} = m \rightarrow \Diamond(\textit{receivedMessageA} = m))$$

Property 2 (Termination)

$$\Diamond(\textit{receivedMessageB} = -1 \land \textit{receivedMessageA} = -1)$$

Property 3 (Causal Consistency)

$$\Box(\textit{receivedMessageB} = m \rightarrow m \in \textit{sentMessagesA})$$

In our PlusCal implementation of the echo protocol we define macros for sending and delivering messages using the specified module interface. By using macros, repetitive tasks such as appending messages to the inbox, checking for available messages, and updating the link structure are simplified. This modular approach ensures that operations align with the underlying communication semantics of the specified link interface, making the implementation more structured and easier to understand. For the Perfect Link we can abstract both *send* and *receive* with the macros:

```
macro send(from, to, msg)begin
    link := Send(link, from, to, msg) ;
end macro ;

macro receive(proc, message)begin
    await HasMessage(link, proc) ;
    message := UnwrapMessage(Message(link, proc)) ;
    link := Receive(link, proc) ;
end macro ;
```

6.1. Echo protocol with Perfect links

Initially, we modeled the echo protocol using the Perfect Link module. In this specification, we were able to verify that all desired properties held true for a given finite number of messages. This implementation served as a baseline to ensure the correctness of the protocol in an ideal communication setting.

6.2. Echo protocol with Fair-loss links

Starting from the Perfect Link implementation, we made necessary modifications to the macro definitions to adapt them to the Fair-Loss structure. Additionally, we introduced non-determinism through the either function, which bifurcates each state into two branches: one where the non-determinism holds true and the message is dropped, and another where the message is not dropped. This approach ensures that every possible message-drop scenario is verified. The macros used for sending and delivering messages are as follows:

```
macro send(from, to, msg)begin
    either
        nonDeterminism := TRUE ;
    or
        nonDeterminism := FALSE ;
    end either ;
    link := Send(link, from, to, msg, nonDeterminism) ;
end macro ;

macro receive(proc, messages)begin
    await HasMessage(link, proc) ;
    with wrappedMessage ∈ Messages(link, proc) do
        link := Receive(link, proc, wrappedMessage) ;
        messages := Append(messages, UnwrapMessage(wrappedMessage)) ;
    end with ;
end macro ;
```

When switching to the Fair-loss Link module, we observed changes in behavior. The inherent unreliability of the Fair-Loss Link made it easy to reach inconsistent states

where properties were violated, particularly when *MaxDrops* was set to a value greater than zero. Specifically, message loss often led to unhandled states, causing deadlocks or incomplete protocol execution. These issues arose even before we could validate any of the desired temporal properties. This experience highlighted the critical importance of designing distributed systems to account for such scenarios, as unreliable communication is a common occurrence in real-world systems.

6.3. Echo protocol with Stubborn links

During the verification of the echo protocol using the Stubborn Link abstraction, we observed a considerable increase in the state space size. This growth in state space can be attributed to the retransmissions used by the Stubborn Link, which ensures reliable message delivery through retransmissions. We set the value of *MaxCopies* to two, as follows:

INSTANCE $StubbornLink$ WITH $MaxCopies \leftarrow 2$

Unlike the execution of the echo protocol with the Fair-Loss Link, which often resulted in inconsistent states or incomplete executions due to message loss, the state space verification of the protocol completed successfully under the Stubborn Link. The retransmission mechanism effectively mitigated the unreliability of message delivery, ensuring that all intended messages were eventually received. However, this came at the cost of increased computational complexity, as the system had to manage multiple redundant messages while maintaining out-of-order delivery semantics. This challenge was particularly evident in our implementation of the echo protocol, where every message sent was forwarded, effectively doubling the duplication overhead.

Due to the duplication behavior, we encountered states where *Property 1* was not valid. For example, in a scenario where process B received a FIN message and subsequently forwarded a duplicated message to process A that was not a FIN, process B could finish execution before the property was satisfied. Similarly, *Property 2* did not hold, as the duplication and out-of-order behavior led to a state where process A received a FIN message while there were still messages in its inbox waiting to be delivered. *Property 3* was the only property that held true, guaranteeing that all received messages were previously sent.

7. Conclusion

In this paper, we introduced a modular and reusable TLA+ library for modeling fundamental communication primitives in distributed systems. Our library includes point-to-point communication over Perfect Links, Fair-Loss Links, and Stubborn Links, providing a structured way to model different reliability assumptions. By encapsulating these communication behaviors, our approach simplifies the specification, analysis, and verification of distributed protocols, reducing the complexity for system designers.

As future work, we aim to extend the library to support multicast and broadcast communication primitives with varying reliability and ordering guarantees. This includes exploring causal, FIFO, and total order delivery semantics. By incorporating these extensions, we seek to further generalize the framework, enabling more comprehensive verification of distributed protocols under diverse communication conditions, including group communication and replicated services.

References

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. The MIT Press.

Braithwaite, S., Buchman, E., Konnov, I. V., Milosevic, Z., Stoilkovska, I., Widder, J., and Zamfir, A. (2020). Formal specification and model checking of the tendermint blockchain synchronization protocol (short paper). In *FMBC@CAV*.

Cachin, C., Guerraoui, R., and Rodrigues, L. (2011). *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer.

dmilstein (2019). *Channels: TLA+ modules for modeling message-passing with different guarantees*. Available at `https://github.com/dmilstein/channels`, accessed: 2025-01-28.

Dotti, F. L., Mendizabal, O. M., and dos Santos, O. M. (2005). Verifying fault-tolerant distributed systems using object-based graph grammars. In *Dependable Computing*, pages 80–100. Springer Berlin Heidelberg.

Gärtner, F. C. (1999). Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26.

Grundmann, M. and Hartenstein, H. (2023). Towards a formal verification of the lightning network with tla+. arXiv:2307.02342 [cs.LO].

House, A. and Tang, P. (2018). A tla+ module for asynchronous message-passing systems. In *SoutheastCon 2018*, pages 1–7.

Kolb, J., Yang, J., Katz, R. H., and Culler, D. E. (2020). Quartz: A framework for engineering secure smart contracts. Technical Report UCB/EECS-2020-178, EECS Department, University of California, Berkeley.

Lamport, L. (1994). The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923.

Lamport, L. (2009). The pluscal algorithm language. In Leucker, M. and Morgan, C., editors, *Theoretical Aspects of Computing - ICTAC 2009*, pages 36–60, Berlin, Heidelberg. Springer Berlin Heidelberg.

Lu, T., Merz, S., and Weidenbach, C. (2011). Towards verification of the pastry protocol using tla+. volume 6722 of *Lecture Notes in Computer Science*, pages 244–258. Springer.

Salamah, S., Gates, A., Roach, S., and Mondragon, O. (2005). Verifying pattern-generated ltl formulas: A case study. In *Model Checking Software, Proceedings of the 12th International SPIN Workshop*, volume 3639 of *Lecture Notes in Computer Science*, pages 200–220. Springer.

Saltzer, J. H., Reed, D. P., and Clark, D. D. (1984). End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288.

Schneider, F. B. (1993). What good are models and what models are good? In *Distributed Systems*, pages 17–26. ACM Press/Addison-Wesley Publishing Co., USA, 2nd edition.

Yin, J.-Q., Zhu, H.-B., and Fei, Y. (2020). Specification and verification of the zab protocol with tla+. *Journal of Computer Science and Technology*, 35:1312–1323.