NetFlex: A Simulation Framework for Networked Control Systems

Katarina Stanojevic[1], Martin Steinberger[1], Maris Siljak[2], Jakob Ludwiger[3] and Martin Horn[1]

Abstract— **This paper presents NetFlex, a MATLAB/ Simulink-based simulation framework for modeling and analysis of networked control systems. Based on a node-based architecture, NetFlex allows users to construct flexible network topologies while integrating customizable control and estimation strategies. By providing an intuitive and adaptable simulation environment, it enables researchers to evaluate the impact of network effects without the complexity of low-level network configurations. As an open-source project, NetFlex promotes collaboration and extensibility, offering a structured framework for studying distributed and communication-constrained control systems. The paper details its design, implementation and capabilities, demonstrating its effectiveness through a case study.**

I. INTRODUCTION

Advancements in wireless networking, cloud computing, and distributed architectures have enabled the development of networked control systems (NCS), where communication networks replace dedicated wiring. While this offers greater flexibility and scalability, it also introduces network effects that must be carefully addressed in control design. In this context, simulation plays a crucial role in developing, testing, and validating control strategies in a safe and cost-effective manner. Beyond analyzing the impact of network-induced effects such as delays, data loss, and aperiodic sampling, simulation enables repeatable testing, which is not feasible in real-world experiments, allowing for direct comparison of control strategies. Moreover, a simulation environment is essential for evaluating different network topologies and comparing communication technologies, as it removes the complexity and effort required for physical implementation.

Simulating NCS requires capturing both dynamics of the considered control system and network-induced effects. Several existing simulation tools address this challenge, each offering different levels of detail in modeling control and communication interactions, for an overview see [1]. MATLAB/Simulink provides a well-established framework for control design and analysis, but it does not inherently support network-induced effects without additional modifications. It lacks proper handling of packet-based communication and variable transmission delays, since standard delay blocks apply delays to sampling instances rather than individual packets, leading to inaccurate representations of network-induced effects, see [2], [3]. MATLAB also does not support arbitrary delay specifications, as existing delay blocks assume transmission delays that are fixed or constrained to multiples of the sampling time, limiting flexibility in modeling real-world communication constraints. Furthermore, additional issue represents packet reordering, which frequently occurs in networked systems due to multi-hop routing or interference, but is not naturally modeled in MATLAB's delay structures. One approach to addressing these limitations is SimEvents [4], a discrete-event simulation tool developed by MathWorks. SimEvents provides event-driven modeling capabilities, allowing users to simulate packet-based communication, queuing mechanisms, and resource contention in networked systems. Unlike MATLAB's standard delay blocks, which operate on sampled data, SimEvents treats messages as individual entities traveling through a network of queues, enabling more accurate modeling of variable transmission delays, congestion effects, and packet reordering. While SimEvents improves network modeling within MATLAB, its focus is primarily on discrete-event scheduling and queuing behavior, making it more suitable for network performance analysis rather than control system validation. Defining control and estimation strategies within a SimEvents framework requires additional effort, as users must configure event-based scheduling policies, which can add complexity when the primary goal is to evaluate control algorithms under network constraints. Furthermore, several simulation tools, such as NS-3 [5], Objective Modular Network Testbed in C++ (OMNeT++) [6] and OPNET are widely used for network simulations, offering detailed models of network protocols, routing, and traffic control, see [1]. However, these tools primarily focus on communication networks. Similarly, Ptolemy II [7] provides a heterogeneous modeling framework that supports co-simulation of control and communication systems, enabling the integration of discrete-event, continuous-time, and hybrid models, but its complex setup and lack of direct MATLAB/Simulink integration make it less practical for control-focused NCS simulations.

One of the most established tools for simulating NCS is a MATLAB/Simulink-based simulator TrueTime proposed in [8], [9]. TrueTime provides framework that enables the co-simulation of control tasks, real-time scheduling policies, and network-induced imperfections. It allows users to model task execution, communication protocols and network delays, making it a powerful tool for investigating the impact of timing constraints, resource allocation, and network effects on control performance. Through its event-driven execution and configurable network models, TrueTime enables the real-time interaction between control and communication layers, offering valuable insights into the behavior of distributed and embedded control systems. However, while

[1] Katarina Stanojevic, Martin Steinberger and Martin Horn are with the Institute of Automation and Control, Graz University of Technology, Graz, Austria. `katarina.stanojevic@tugraz.at`.

[2] Maris Siljak is with DiLT Analytics, Graz, Austria.

[3] Jakob Ludwiger was with the Institute of Automation and Control, Graz University of Technology, Graz, Austria.

extremely powerful and highly detailed in modeling real-time execution, TrueTime primarily focuses on communication and scheduling aspects, which may introduce unnecessary complexity when evaluating control strategies. Features such as detailed medium access control modeling and low-level network protocols are highly relevant for network engineers, but often exceed the level of detail needed for control design. Researchers developing and testing new controllers typically need to evaluate performance under simplified network assumptions—such as delays, dropouts, or message reordering—without investing significant time in configuring task scheduling and communication models.

To address these challenges, this paper presents NetFlex, a MATLAB/Simulink-based simulation framework designed to bridge the gap between control and communication in NCS simulations. NetFlex builds upon TrueTime, leveraging its powerful Simulink blocks and core functions while introducing an additional abstraction layer that simplifies the simulation process. This layer provides a structured interface between the user and TrueTime, simplifying the configuration of network effects without requiring in-depth knowledge of real-time task execution or network communication handling. Therefore, by maintaining TrueTime's core functionality but enhancing accessibility, NetFlex shifts the emphasis toward control strategy validation, enabling researchers to test control strategies under realistic yet configurable network conditions without the need for extensive setup of scheduling policies or detailed communication models. To achieve this, NetFlex is designed to provide an intuitive platform that adopts a highly adaptable, node-based architecture. This allows users to construct simulations by assembling predefined network nodes, which represent both core system components—such as sensors, controllers and observers—and dedicated network-effect nodes, such as delay nodes for transmission delays and dropout nodes for data loss. Each node can be easily configured and extended, allowing seamless integration of user-defined control and estimation strategies without modifying the simulation framework. The entire network structure and interactions between nodes are fully user-defined, enabling users to construct diverse network topologies, including distributed sensing networks and spatially distributed controllers. Furthermore, by allowing precise placement of network effects and their individual configuration, the framework facilitates a detailed analysis of the impact of communication constraints on system performance. Furthermore, as a highly modular and extensible open-source simulation environment, NetFlex encourages collaboration and knowledge sharing within the research community. The framework is publicly available on GitHub[1], allowing researchers to access, modify, and extend its functionality to suit their specific needs.

This paper provides an overview of NetFlex's design and implementation, highlights its integration with TrueTime, and demonstrates its capabilities through a case study, showcasing its potential for advancing the study of NCS.

[1]`https://github.com/katarinas37/NetFlex`

II. Simulation Framework: Object-Oriented Design and Features

The NetFlex framework follows an object-oriented design, implementing NCS components as MATLAB classes and Simulink TrueTime based blocks. The overall structure of these components is illustrated in the unified modeling language (UML) class diagram shown in Fig. 1.

A. Incorporating Network Nodes

At the core of this design is the `NetworkNode` class, which serves as the foundation for various node types. As an abstract class, it cannot be instantiated directly but instead provides a standardized interface for defining NCS components. Each `NetworkNode` instance is defined by its number of inputs and outputs, the nodes it communicates with, and a unique identifier, allowing users to flexibly define network topologies by specifying how nodes interact. Additionally, it introduces a structured approach to initializing and scheduling networked tasks. The abstract method `init()` must be implemented by each subclass to define the node's behavior, including configuring the TrueTime kernel, where network-related tasks are created and scheduled. To support task execution, a wrapper function accepts a function handle, enabling the creation and scheduling of TrueTime tasks that invoke methods from the calling class instance.

As shown in Fig 1, multiple node types inherit from `NetworkNode`. These can be grouped into three categories: nodes modeling network effects, nodes for control and observer components and specialized supporting nodes providing functionalities such as sensor node or additional mechanisms, e.g. buffering or message rejection. Each of these subclasses defines its own functionality while adhering to the standardized communication and task scheduling framework established by `NetworkNode`. Thereby, the consistency in network interactions is ensured, while allowing specialized behavior of the nodes tailored to their specific roles.

1) Network-Induced Effects: Delay and Dropout Nodes

The `VariableDelay` class inherits from `handle` and `NetworkNode`, making it a reference-based class. It provides a framework for variable delays, defining how messages are received, processed and transmitted. `VariableDelay` relies on several TrueTime functionalities, see Table 1 in [8]. It uses `ttCreateTask` to define two tasks: a delay task, determining the transmission time, and a send task, forwarding the message when the delay expires. Messages are received and sent using `ttGetMsg` and `ttSendMsg`. The TrueTime kernel is initialized with `ttInitKernel('prioDM')`, enabling deadline-monotonic scheduling, where tasks with shorter deadlines are prioritized. Finally, `ttSleepUntil` ensures messages remain buffered until scheduled transmission time.

The delay dynamics in `VariableDelay` relies on a buffer to ensure that each received message is held for a computed duration before forwardin it to the next node. This is managed by the `MsgBuffer` class, which stores messages as `BufferElement` objects, see Fig. 1. Each `BufferElement` contains data packet along with a spec-

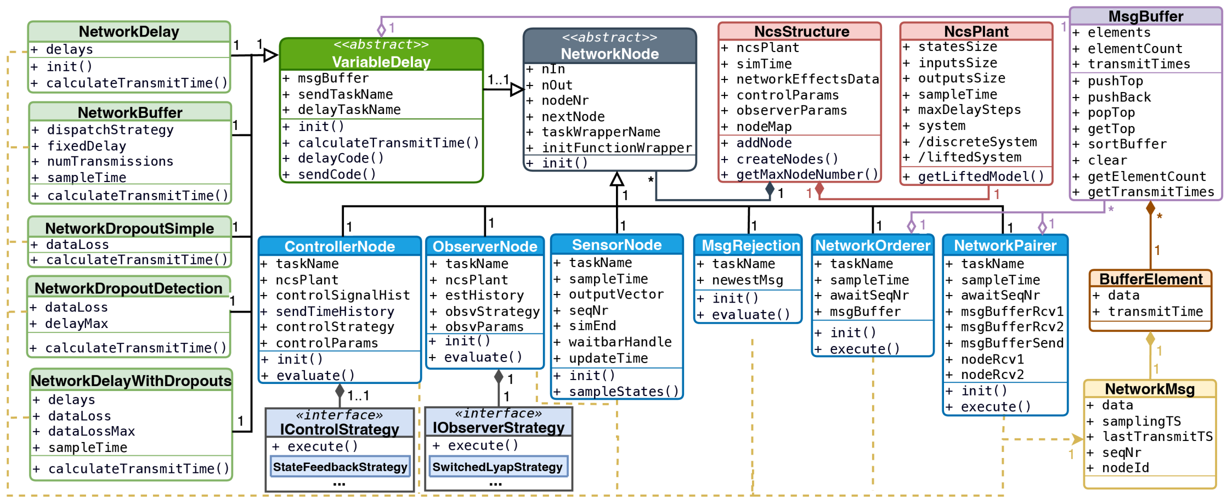

Fig. 1. UML class diagram of the NetFlex Framework, illustrating the key components and their interactions within a NCS simulation framework.

ified transmit time when the message should be released. The buffer supports operations such as adding, removing, and sorting messages based on their transmit times, ensuring correct message sequencing and delay simulation. The transmission time is determined by the abstract method `calculateTransmitTime()`, allowing subclasses to define their own delay computation strategies. A key use case of `VariableDelay` is to model different types of network delays, which can be manually defined, statistically modeled, or extracted from real-world data. This functionality is implemented in the subclass `NetworkDelay`, see Fig. 1. Additionally, this class serves as the foundation for other NCS components, such as a network buffer [10], [11], which introduces additional delays based on message timestamps, as implemented in `NetworkBuffer`, see Fig. 1.

Beyond modeling delays, data loss can be handled in a simple yet effective way by assigning a very large delay to a message, preventing its transmission within the simulation time frame. This approach allows packet dropouts to be managed within the same framework as variable delays. By setting an excessively high transmission time, the message remains buffered indefinitely, mimicking the effect of data loss, as realized in the `NetworkDropoutSimple` subclass depicted in the Fig. 1. If explicit data loss detection is required, as in [12], it can be incorporated by, e.g., introducing flags that are sent once the maximum allowable delay has passed, as implemented in the `NetworkDropoutDetection`, see Fig. 1. Nevertheless, this approach can be further extended to incorporate more advanced solutions, such as e.g. adaptive delay adjustments, allowing for a more detailed representation of network reliability.

Sequences for both network delays and data loss can be generated in the framework following the specified rule or loaded from pre-generated or real-world data. Each sequence is structured so, that the k-th element defines the transmission status or delay of the data packet with sequence number k. These are provided to the nodes, applying the specified network effects consistently throughout the simulation.

2) *Controller and Observer Nodes*

The class `ControllerNode` serves as an abstract base class for implementing various control strategies within the framework, which utilizes a unified interface `IControlStrategy` while allowing for flexibility in designing specific control laws. It provides methods such as `init()`, which configures the node and its parameters and `evaluate()`, which manages communication with other nodes. Concrete control strategy is realized through interface extended by, e.g., `StateFeedbackStrategy`, shown in Fig. 1 for illustrative purposes, which implements a state-feedback control law in respective `execute()`.

Similarly, the `ObserverNode` class establishes a common structure for integration of different estimation techniques. For instance, the `SwitchedLyapStrategy` in Fig. 1 extends `IObserverStrategy` interface by implementing a switched Lyapunov-based observer from [12].

3) *Supporting Nodes*

The `SensorNode` is required to generate and transmit measurements. In addition, other specialized nodes can be implemented if required by the chosen strategy, e.g. `MsgRejection` node, which filters and discards outdated or invalid packets or `NetworkOrderer` node, which ensures that received data is processed in the correct order.

B. Message Handling and Data Exchange

The data transmission is handled using the `NetworkMsg` class, which provides a structured format for messages exchanged between network nodes. Each `NetworkMsg` object encapsulates all essential communication information, including the data, corresponding sampling and last transmission timestamps, and a sequence number to track message order. The `NetworkMsg` object is created within the `SensorNode` class, which runs a TrueTime task executing at each sampling step to obtain the current system state. The data is then encapsulated in a `NetworkMsg` object and sent through the NCS. Each receiving node processes the message according to user-defined operations, which can involve state estimation, control computation, or network-induced transformations such as delays and dropouts.

C. Problem Formulation and NCS Structure

To establish the problem formulation and system structure while ensuring a consistent and flexible representation of NCSs, the framework introduces two key components: the `NcsPlant` and `NcsStructure` classes. The class `NcsPlant` provides a structured way to define the system dynamics, as it stores all essential parameters, e.g. the continuous-time state-space model of the plant and the sampling time. This ensures that all simulation components have access to the necessary information. Additionally, discrete-time and lifted system models, [13], are derived within `NcsPlant`, enabling a seamless integration of control strategies. `NcsStructure` determines the NCS topology and data flow between nodes. It automatically generates all necessary components for the simulation, ensuring that each node is correctly instantiated and connected. Additionally, `NcsStructure` defines the network effects which are used in the individual delays and data loss network nodes.

D. Simulation Execution and Post-Processing

To set up a simulation, users first instantiate an `NcsStructure` object, which defines all NCS components, thereby creating all the required nodes and storing them as a property of this object. In Simulink, users simply place the provided TrueTime Kernel based blocks and enter the corresponding node reference. The framework automatically links each block to the predefined node, ensuring that all execution logic is applied without requiring additional configuration, which completely eliminates the need for manual configuration. Once the simulation is completed, the results are automatically stored for post-processing in the member variable `results` of the `NcsStructure` object.

III. Framework Application Example

To demonstrate the performance and flexibility of the developed framework, the following example is considered. The system under study is deliberately kept simple, as the focus of this paper is on the framework itself rather than the specific methods used. It is a second-order linear time-invariant system with the state-space representation, given by $\dot{\boldsymbol{x}}(t) = \boldsymbol{A}_c\boldsymbol{x}(t) + \boldsymbol{b}_c\hat{u}^*(t) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \boldsymbol{x}(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \hat{u}^*(t)$ and $y(t) = \boldsymbol{c}_c^T \boldsymbol{x}(t) = \begin{bmatrix} 1 & 0 \end{bmatrix} \boldsymbol{x}(t)$ with a constant sampling time $T_d = 5$ms. Since not all system states are measurable, an observer is required to estimate the unmeasured states. The strategy chosen for this study follows a switched Lyapunov function-based approach from [12], which compensates for data loss by leveraging the most recently available output measurements while maintaining the separation principle. The Luenberger-inspired observer gain is designed using a linear matrix inequality (LMI)-based formulation, where different gains are applied depending on the number of consecutive packet dropouts, ensuring stability and computation of state estimates despite unpredictable data availability.

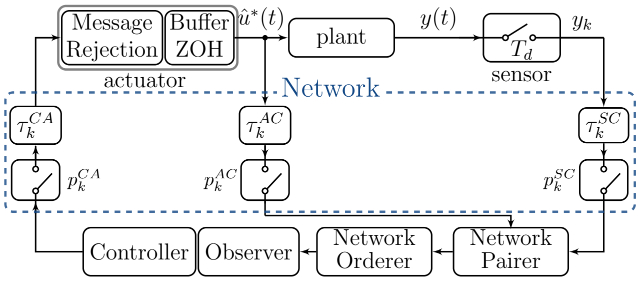

Fig. 2. Simulation example: considered NCS architecture

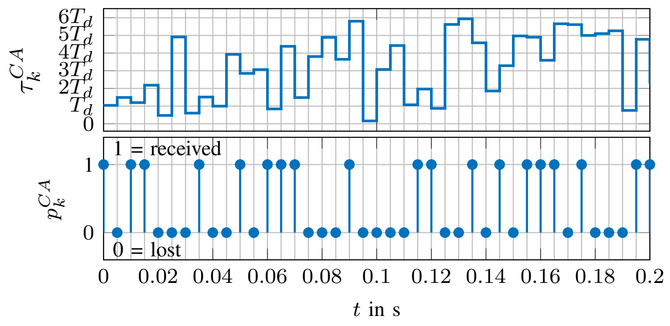

Fig. 3. Simulation Example: Network effects in the communication path from the controller to the actuator

The control law follows a state-feedback design based on a switched system representation from [11], with control gains obtained via low-complexity LMI conditions.

This strategy necessitates a specific NCS structure given in Fig. 2, see [12], where three communication paths over the network are required, including the transmission of the control signal to a centralized controller. Each communication path is subject to network effects, where delays are represented by τ_k^*, denoting the time-varying transmission latency, and data loss is represented by p_k, indicating whether a packet is received ($p_k = 1$) or lost ($p_k = 0$). The subscripts $*$ specify the respective communication paths: SC (sensor-to-controller), AC (actuator-to-controller), and CA (controller-to-actuator). The methods from [11], [12] are suitable for NCS in which network effects can be modeled by a maximum allowable bound (MAB), which defines strict upper limits on delays and packet dropouts rather than relying on statistical distributions. To generate the considered scenario, these MABs are used to introduce time-varying network effects while ensuring that delays remain within the specified bounds for this example, $\bar{\tau}^{AC} = 4T_d, \bar{\tau}^{SC} = 5T_d, \bar{\tau}^{CA} = 6T_d$, and that number of subsequent packet dropouts does not exceed $\bar{p}^{AC} = 2$, $\bar{p}^{SC} = 3$ and $\bar{p}^{CA} = 4$. For illustrative purposes, Fig. 3 presents a representative 0.2-second segment to clearly visualize the network effects for a single communication channel, from the controller to the actuator. Please note, that the assumption of MAB is made solely because the chosen NCS structure, selected for illustrative purposes, follows [12]. In general, the framework allows for full flexibility in defining network effects, enabling users to specify them according to their own requirements.

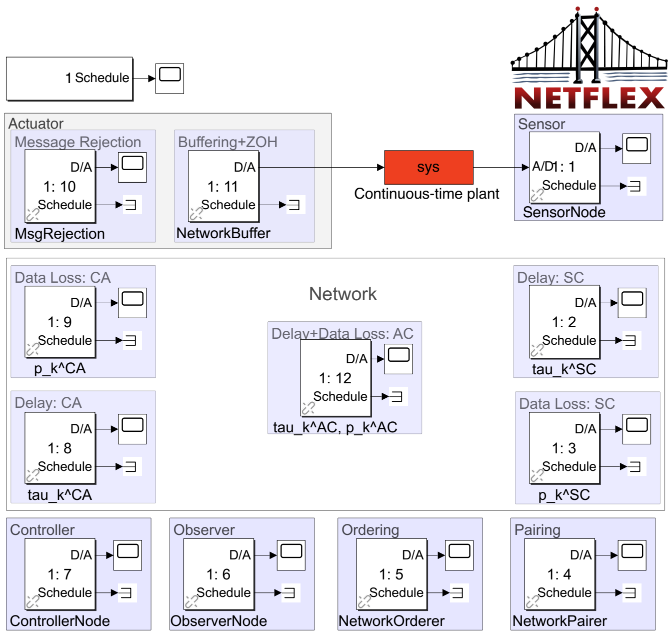

Fig. 4. Simulation example: Simulink model

To model data loss, three different blocks are implemented, each handling packet dropouts according to the specific requirements of the communication path. For the sensor-to-controller path, the method from [12] requires data loss detection at the controller side. Therefore, in this path, the data loss block includes a flagging mechanism, where a flag is transmitted after the maximum possible delay has elapsed to indicate that a packet was lost. In the actuator-to-controller path, data loss and delay are implemented together within a single network node due to the requirements of [12]. Here, an adaptive mechanism is used: if data loss occurs, the node waits for the next control signal to arrive at the controller. The lost packet's transmission time is set to match that of the newly received packet, ensuring compatibility with the approach in [12]. For the controller-to-plant path, no additional mechanisms are required by [14]. Data loss is modeled as a large delay, effectively preventing the lost packet from being received within the simulation time frame. For implementing network delays in the sensor-to-controller and controller-to-actuator paths, `NetworkDelay` blocks are used, which determine the transmission time by adding the delay to the time instant at which the packet is received.

Furthermore, the method from [12] requires multiple dedicated mechanisms to implement the proposed strategy, as shown in Fig. 2, each of which can be realized as a separate node within NetFlex. First, `NetworkPairer` is responsible for pairing the corresponding control and output signals, which can be implemented based on their original transmission timestamps. Once paired, `NetworkOrderer` ensures that the data packets are correctly arranged before being forwarded to the `ObserverNode`, where the state observer is implemented. The `ControllerNode` operates as a separate entity, computing the control law based on the predicted system states provided by the observer. Lastly, an additional buffering mechanism is required at the actuator side, which ensures that the applied control signal is updated only at regular intervals synchronized with the sample time T_d, including a Zero-Order Hold (ZOH) to generate a continuous control input out of the discrete time sequence. This functionality is realized through the `NetworkBuffer`, whose output is provided to the plant. The resulting Simulink model incorporating the used observer-based control strategy along with network effects, is shown in Fig. 4, which closely follows this theoretical network architecture given in Fig. 2 in a clear and modular way, making the transition from theory to implementation straightforward. Its structured design ensures that key components—such as user-defined strategies and handling of network delays and data loss—are easily configurable and adaptable, allowing for seamless modifications without altering the core architecture.

For the specified network configuration, the following observer and controller gains are obtained: $\boldsymbol{k}^T = \left[0.057, 0.289\right]^T$, $\boldsymbol{l}_0^T = \left[0.661, 9.51\right]^T$, $\boldsymbol{l}_1^T = \left[0.176, 2.56\right]^T$, $\boldsymbol{l}_2^T = \left[0.117, 1.51\right]^T$, $\boldsymbol{l}_3^T = \left[0.0925, 0.939\right]^T$, see [11], [12] for details. Fig. 5 shows the evolution of the system states which exhibit asymptotic convergence to zero, included for completeness, to confirm that the framework correctly simulates the expected system behavior. Fig. 6 illustrates how the control signal evolves as it propagates through the NCS, demonstrating the correct functionality of the developed simulation framework. The figure presents the control signal only at the selected network nodes in a 0.2-second segment to keep the visualization clear. Additionally, to maintain readability, the grid in the x-direction is aligned with the sampling period, i.e. placing grid lines at $t = kT_d, k \in \mathbb{N}_0$. After pairing and ordering, state estimates are provided to the controller with a delay but without data loss, since the strategy compensates for missing packets, as discussed in [12]. The `ControllerNode` computes the control law based on these delayed state estimates. The black line in Fig. 6 represents an auxiliary variable that visualizes the control signal as if no network effects were present. Please note, that although this signal is generated based on state estimates that have already experienced network delays and data loss, this is not an actual signal occurring in the NCS. This signal can be reconstructed only in post-processing by using the computed control data and aligning it with $t = kT_d$, disregarding presence of network effects. This allows for a clearer illustration of how the control input evolves over time and facilitates comparison with the ideal case. The blue line represents the actual control signal sent by the controller, plotted with the correct transmission timing. In some instances, multiple values appear at the same time instant, which occurs when packets arrive at the controller simultaneously due to reordering, which happens when newer packets must wait for older ones in the `NetworkOrderer` before all being forwarded at the same time. The blue signal is transmitted over the network, where it experiences both delay τ_k^{CA} and data loss p_k^{CA}. The red line represents the

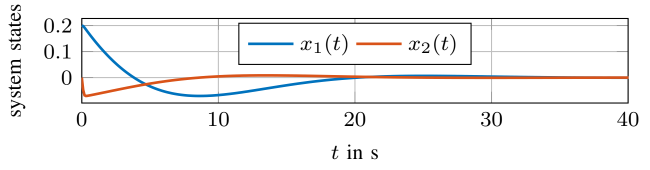

Fig. 5. Evolution of the system states $\boldsymbol{x}(t)$.

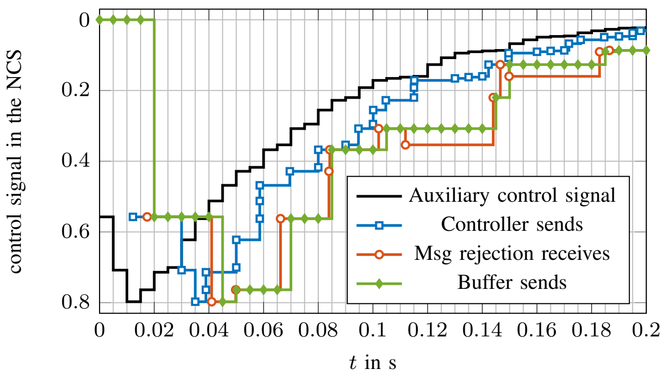

Fig. 6. Propagation of the control signals in the networked system

control signal received by the actuator node, which differs from the blue line due to missing control updates caused by data loss. Specifically, at time steps where packet loss occurs in Fig. 3, the red line holds its previous value, whereas the blue line updates its value. Furthermore, network-induced delays introduce a time-varying misalignment between the blue and red signals, shifting control updates forward in time according to the delays shown in Fig. 3. Additionally, it is evident that the order of received data is not always preserved, as out-of-order packet arrivals frequently occur in scenarios with large delays. Before forwarding to the buffer, older data is discarded by the message rejection mechanism, ensuring that only the most recent data is processed. As a result, certain points from the red line are missing in the green buffer line, reflecting the rejection of outdated control signals. Additionally, the buffer updates the control input only at the start of each sampling period, i.e., at $t = kT_d$, using the most recent received value, which is evident in the Fig. 3, as the buffer actuation points perfectly align with the sampling grid. If no new data arrives, the previously applied control input remains active, as defined by the strategy.

IV. Summary and Outlook

This paper presents a modular MATLAB/Simulink-based simulation framework for NCS which builds upon True-Time's capabilities but introduces an additional abstraction layer, allowing researchers to configure and test control strategies more efficiently and intuitively. Through the structured inheritance hierarchy, the NetFlex framework enables a modular and extensible representation of NCS, where new nodes can be introduced with minimal effort while maintaining compatibility with the simulation framework. It not only enables flexible modeling of communication effects but also allows users to incorporate custom strategies such as message rejection, buffering, and data packet reordering, ensuring a detailed analysis of their impact on control performance. As a result, it can simulate a wide range of network topologies, protocol and control scenarios, making it a valuable open-access tool for researchers in evaluating the robustness of control strategies under realistic network conditions.

A case study demonstrated the functionality and adaptability of the framework, illustrating how the observer-based control strategy from [14], chosen for illustrative purposes, performs under network-induced uncertainties. The simulation results confirmed the accurate modeling of transmission delays and data loss, as well as the effectiveness and flexibility of the implemented strategy-specific mechanisms.

Future work may focus on further expanding the framework's capabilities in two key areas: integrating real-time systems for hardware-in-the-loop simulations to enable practical performance validation and enhancing the user experience with a graphical interface for intuitive configuration and scenario definition.

References

[1] M. Sollfrank, M. von Freymann, and B. Vogel-Heuser, "Overview and classification of approaches for the simulation of networked control systems," *at - Automatisierungstechnik*, vol. 68, no. 3, pp. 151–165, 2020.

[2] F. Zhang and M. Yeddanapudi, "Modeling and simulation of time-varying delays," in *Spring Simulation Multiconference*, 2012.

[3] M. Steinberger, M. Tranninger, M. Horn, and K. H. Johansson, "How to simulate networked control systems with variable time delays?" *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 3098–3103, 2020, 21st IFAC World Congress.

[4] X. Xu and Z. Wang, "Networked modeling and simulation based on simevents," in *2008 Asia Simulation Conference - 7th International Conference on System Simulation and Scientific Computing*, 2008, pp. 1421–1424.

[5] A. Mohta, S. Ajankar, and M. Chandane, "Network simulator-3: A review," 08 2023.

[6] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment," in *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, ser. Simutools '08, 2008.

[7] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[8] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Arzen, "How does control timing affect performance? analysis and simulation of timing using jitterbug and truetime," *IEEE Control Systems Magazine*, vol. 23, no. 3, pp. 16–30, 2003.

[9] M. Andersson, D. Henriksson, A. Cervin, and K. Arzen, "Simulation of wireless networked control systems," 01 2006, pp. 476 – 481.

[10] K. Stanojevic, M. Steinberger, J. Ludwiger, and M. Horn, "Predictive multivariable sliding mode control of buffered networked systems," in *2022 European Control Conference (ECC)*, 2022, pp. 981–986.

[11] K. Stanojevic, M. Steinberger, and M. Horn, "Switched lyapunov function-based controller synthesis for networked control systems: A computationally inexpensive approach," *IEEE Control Systems Letters*, vol. 7, pp. 2023–2028, 2023.

[12] K. Stanojevic, M. Steinberger, and M. Horn, "Robust state estimation in networked control systems under data loss and delays: A switched lyapunov funtion based approach," in *2025 European Control Conference (ECC)*, 2025.

[13] M. Cloosterman, L. Hetel, N. van de Wouw, W. Heemels, J. Daafouz, and H. Nijmeijer, "Controller synthesis for networked control systems," *Automatica*, vol. 46, no. 10, pp. 1584 – 1594, 2010.

[14] K. Stanojevic, M. Steinberger, and M. Horn, "State estimation in networked control systems with time-varying delays: A simple yet powerful observer framework," in *2024 IEEE 63rd Conference on Decision and Control (CDC)*, 2024, pp. 6916–6921.