
© Copyright 2009 IEEE CS Press
Hybrid System Simulation of Computer Control Applications over Communication Networks

Guosong Tian, Colin Fidge and Yu-Chu Tian
School of Information Technology
Queensland University of Technology
Brisbane QLD 4001, Australia
{g.tian, c.fidge, y.tian}@qut.edu.au

Abstract—Discrete event-driven simulations of digital communication networks have been used widely. However, it is difficult to use a network simulator to simulate a hybrid system in which some objects are not discrete event-driven but are continuous time-driven. A networked control system (NCS) is such an application, in which physical process dynamics are continuous by nature. We have designed and implemented a hybrid simulation environment which effectively integrates models of continuous-time plant processes and discrete-event communication networks by extending the open source network simulator NS-2. To do this a synchronisation mechanism was developed to connect a continuous plant simulation with a discrete network simulation. Furthermore, for evaluating co-design approaches in an NCS environment, a piggybacking method was adopted to allow the control period to be adjusted during simulations. The effectiveness of the technique is demonstrated through case studies which simulate a networked control scenario in which the communication and control system properties are defined explicitly.

I. INTRODUCTION

A networked control system (NCS) consists of a number of physical and computational elements, or agents, which have both physical and informational interactions and dependencies, supported by a common networking infrastructure [1]. Although principles for feedback-driven control of physical processes have been explored thoroughly in the literature, the insertion of a digital communications network with variable message propagation characteristics into the feedback cycle makes the design and analysis of a networked control system complex and challenging, especially for complex and large-scale networks [2], [3].

Traditionally, the design of the communications network and the interacting control components it connects have been treated separately when the dynamic behaviour of an NCS was analysed. However, in recent NCS research, co-design methodologies have been developed, where the design of control systems takes network resource constraints into account, and where real-time computing and scheduling, and even message routing protocols, are designed with end-to-end control performance in mind [4], [5], [6], [7]. This means that networking issues, such as bandwidth and message delays, must be considered simultaneously with control system issues, such as stability, performance, fault tolerance and adaptability. This presents a challenge for conventional simulation tools which are thus required to explicitly model, measure and analyse both networking and control performance.

Work in the area of measurement and verification of control and network performance of an NCS has generally proceeded along three paths: theoretical investigation, experimental testbeds, and simulation. Theoretically modelling an NCS accurately is very important. However, modelling the behaviour of these communications networks is complicated and does not scale well to increases in the size of the network and the complexity of modern communication technologies such as WiMax. The second path is to build an NCS testbed. This can more closely mimic the actual operational stresses which control systems and network implementations might face, but constructing such testbeds is labour intensive and costly [8]. The third option, simulation, represents the most practical and easy-to-implement methodology available to control system developers for design and evaluation of both network and control performance of an NCS. Several simulation methodologies (also known as ‘co-simulations’) have been developed, which support simulating communications networks and control systems simultaneously [9], [10] but so far have been limited to specific NCS architectures.

In this paper a general-purpose simulation environment is described which can be used to measure and evaluate both communication and control performance of an NCS, particularly for evaluating a co-design methodology. In order to support the applications at hand, it must be possible to simultaneously simulate the computations that take place within the network nodes, the wired and wireless communication between the nodes, the sensor and actuator dynamics, and the behaviour of the NCS’s environment, including the physical systems being controlled.

This paper describes the design and implementation of a hybrid NCS simulation environment based on NS-2, a general-purpose Network Simulator [11], by integrating continuous-time plant process simulation into a discrete event-driven network simulation. The resulting simulation environment developed can be considered a hybrid between a discrete event-driven network simulator and a time-driven continuous system simulator. In our approach, continuous-time system dynamics are simulated in parallel with discrete network events. For this purpose, a synchronisation method for integrating simulation of continuous-time process dynamics into the discrete network
simulator was developed. A piggybacking method for varying the sampling period is also designed to make it possible to change the control system's sampling period dynamically during the simulations. In addition, the efficiency and accuracy of the resulting simulations are considered.

II. RELATED WORK

Our research involves simulating a networked control system, so in this section we summarise the requirements for such simulations, review the available options for a simulator that can meet these needs, and briefly introduce the particular simulator we used as a starting point.

A. Requirements for NCS Modelling and Simulation

Conventional feedback-driven control theories and controller design methodologies make many idealised assumptions, such as equidistant sensor sampling times, synchronised control, and instantaneous sensing and actuation. These assumptions are invalid for networked control, which raises the issue of how to measure and analyse the impact of network behaviour on NCS performance. Another difficult task in NCS design is validation of network methodologies (such as the playback mechanism in networked control) and co-design methodologies (such as adaptive control based on network measurement) [12], [13]. These problems motivate our work.

Although in previous feedback-driven control theories and discrete-events system models have been proposed [14], [15], no single simulation environment has been available to allow standardised, reproducible and directly-comparable experiments in NCS research. Therefore, a simulation environment that can simulate continuous-time system dynamics and discrete-event communication networks simultaneously is crucial for evaluating potential NCS systems. A range of desirable capabilities for an NCS simulation environment has been identified in the literature, including simulation accuracy of quality-of-control and quality-of-service measurements; easy configuration of continuous processes and network scenarios with high simulation efficiency; and support for simulating a wide range of continuous processes and communication networks such as WiMax [16], [6], [17].

B. Options for Simulating a Networked Control System

To develop a cost-effective simulator for an NCS, an obvious starting point is to begin with an existing simulator. Mathematical modelling and analysis of control system dynamics have been investigated for many years using tools such as MATLAB [18]. More recently, simulators have emerged specifically for analysing communications network behaviours [11]. Given these starting points, there are three possible ways to construct a simulator for NCSs.

1) Building a Network Block for a Control System Simulator: This is the most widely-used simulation method in recent research on networked control. A typical example is TrueTime [9], which is based on Simulink, the graphical simulation environment of MATLAB. As a continuous-time system simulation tool, MATLAB offers powerful modelling and simulation capabilities. A TrueTime block can interact with ordinary Simulink blocks to form a complete NCS model. Unfortunately, previous research using this simulation method has two significant drawbacks that inhibit its application. Firstly, the communication models are often too simplistic to simulate complex communication networks precisely. Current network blocks are implemented based on statistical models which assume that network characteristics follow given probability distributions and that each event is independent [16]. However, this 'average-case' behaviour can be a poor approximation of real-world communication networks in which events tend to be 'bursty' or time-correlated. Secondly, the network block may not provide support for simulating newly emerging communication models, such as WiMax. In the case of TrueTime, for instance, although Simulink has been extended with some transport delay blocks, it remains difficult to define and implement precisely the execution times, start times, finish times, deadlines, etc. of the system tasks [19].

2) Developing an Interface Between a System Simulator and a Network Simulator: In this approach two separate simulators are connected via a newly-developed interface. However, the resulting system is fragile and difficult to update when simulations requiring different information to be passed across the interface are constructed.

For example, by using the external system API in OPNET, Harding designed an interface between MATLAB and OPNET, for constructing a simulation of an NCS based on MANETs (Mobile Ad Hoc Networks) [20]. The interface can be used to pass dynamic system information, from an external simulation performed in MATLAB/Simulink, into an OPNET network simulation. However, this rigid simulation environment makes it difficult to design and evaluate new NCS alternatives under different operating settings. Another issue is that the separation between the control system and network simulations makes it difficult to model co-design methods in which the settings of both the network and the control system interact dynamically.

Another example of this approach is an interface created between network simulator NS-2 and Arena [21], a multirobot simulation tool. However, for a real-time simulation tool it may be difficult to make all of the simulated components in the system fast enough to keep up with real-time events, particularly when simulating a large number of agents.

3) Simulating Continuous-Time Process Dynamics in a Network Simulator: The third approach uses agents to describe system dynamics and integrates agent blocks into a network simulator. An agent represents the properties of network nodes, including the continuous-time dynamics of the physical process being controlled as a 'plant node'. This approach is becoming popular because of its ease of implementation and the accuracy and efficiency of its simulations.

Most network simulators are event driven. They simulate distinct agents that connect together to form a network. They also support a programming or scripting language to allow node behaviours to be defined procedurally. One example is BARAKA, an integrated Sensor/Actuator Network
Sampling event emulating an NCS in laboratory environments [8]. The NSE has been used to implement a platform for emulation supported via a network simulator emulator. Also, network protocols, including protocols for heterogeneous industrial models of continuous systems, can be implemented in C++ and then be combined with Tcl scripts to simulate an NCS. Our software implements the simulation primitives can be implemented in a compiled language (C++) while the core set of high-performance simulation for plant agents, by using an Ordinary Differential Equation (ODE) solver, and packet-level network simulations for communication [10]. However, this framework lacks support for high-order system simulation and full synchronisation between the event-driven network simulation and continuous-time systems simulation. Our work aims to add this flexibility while considering both simulation accuracy and efficiency.

C. The NS-2 Simulator

NS-2 is a discrete event-driven simulator and a widely accepted de facto standard for simulating networks [11]. As a basis for simulating an NCS, using the approach described in Section II-B.3, NS-2 has several advantages. For instance, its class hierarchy makes it easy to implement an interface that lets other programs access, measure, and even manipulate network information or network settings.

In addition, NS-2’s programming model makes it easy to configure networks using a flexible and interactive scripting language (Tcl) while the core set of high-performance simulation primitives can be implemented in a compiled language (C++) for better simulation efficiency. For our purposes, simulation objects can be flexibly implemented in C++ and then be combined with Tcl scripts to simulate an NCS scenario. The C++ part comprises a scheduler and a variety of network components. The Tcl part consists of libraries that give access to the C++ objects. Our software implements the models of continuous systems the same way.

Lastly, NS-2 supports simulation of many ready-to-use protocols, including protocols for heterogeneous industrial networks such as CAN and CANopen [23]. Also, network emulation is supported via a network simulator emulator (NSE). The NSE has been used to implement a platform for emulating an NCS in laboratory environments [8].

III. ARCHITECTURE OF OUR HYBRID SIMULATION

To simulate an NCS, we adopted the third simulation alternative mentioned above (Section II-B). The existing network simulator NS-2 was extended with ‘plant’ and ‘controller’ simulation agents as shown in Fig. 1. To support an NCS simulation, the simulator has to model both digital network events and continuous environmental processes.

As per conventional control system theory [24], the ‘plant agent’ in Fig. 1 can be viewed as a combination of a sensor and an actuator in a control loop. The time cost of the computation of the control algorithm is negligible here.

By use of ODE solvers, the plant agent simulates a real continuous-time plant process with time-varying plant status. Unfortunately, as a discrete-event-driven simulator, NS-2 does not support simulating network and continuous plant processes in true parallelism. As shown in Fig. 2, the events via which the plant and network interact are sensor sampling (which allows the controller to observe the plant’s state) and actuator signal updating (which allows the controller to influence the plant’s state). The simulation therefore needs to update the values of the plant status at the time of these events (sampling and actuation) to model the network and plant process evolving simultaneously. However, the precise timing of network events is not predictable and scheduling events in the past is impossible in NS-2, so we needed to devise an appropriate synchronisation mechanism for sampling events triggered by the continuous-time plant process.

Below we describe further the new components that were added to NS-2 to model an NCS. The scheduling of network events for network simulation and synchronisation events for simulating continuous plant process are also discussed. Further detail on the related simulation methodology, such as the synchronisation mechanism, is then described in Section IV.

A. Plant and Controller Agents in NS-2

In NS-2, an agent is a model of a real object. For our purpose a physical plant process to be controlled, and control algorithms, can be implemented by adding new agent objects. In the implementation, the Agent class has
to be extended with derived classes \textit{PlantAgent} and \textit{ControllerAgent}. A plant agent represents a combination of a sensor and an actuator. Plant process dynamics and control algorithms are programmed as attributes of these two classes.

The scheduler contains a queue of events ready to be executed, ordered by their (simulated) time of occurrence. Transmission of a data packet is treated as an event handled by the scheduler. In NS-2, the data packet’s content is not modelled by the communications event, so sensor samples and actuation signals in an NCS need to be contained in the header specification of the packet and a new packet header class \texttt{NcsHeaderClass} is defined and implemented, which is derived from an existing NS-2 class \texttt{PacketHeaderClass}. The definition of the new packet header class is similar to Liberatore et al.’s work [10]. Once the new classes have been written, they become part of the compiled NS-2 C++ hierarchy. Instances of the new agent classes are the end-points of wired and wireless connections in the NS-2 simulation, and are thus able to pack and insert sampling and control messages into the simulated network.

\textbf{B. Modelling Continuous-Time Plant Processes}

It is well-known that a continuous-time plant can be modelled using numerical analysis methods such as ordinary differential equations (ODE) [25]. Also the status of a continuous-time plant process can be computed if the model and the plant input are known in advance. The plant’s status is changed only if the plant’s input is updated.

In our approach, we use several ODE solvers for first-order and higher-order process models such as Euler’s method and the (adaptive) Runge-Kutta method [26]. To add a new continuous-time process model for a plant agent object, a process model with an ODE solver is programmed as a function of the \texttt{PlantAgent} class. For example, function \texttt{ODE2-RK4} represents a second-order model with an ODE solver using the Runge-Kutta method (order 4). The plant’s status is stored as a set of state variables defined in the \texttt{PlantAgent} class.

Currently, the function library for simulating process models in our approach is yet to be extended. Fortunately, a lot of work has already been done elsewhere on continuous-time system simulation. MATLAB is one of most popular continuous system simulation tools and the MATLAB Compiler Runtime (MCR) supports sharing MATLAB functions in Linux environments [18]. MCR makes it straightforward to add new functions for simulating process models in NS-2, by use of MATLAB function calls in C++ subroutines. The general process is to write MATLAB functions, compile them to a shared library using the MATLAB compiler, and then access this library using C++ code in NS-2 [27].

\textbf{C. Scheduling Synchronisation Events During Simulation}

We have extended the NS-2 package with separate agents to represent plants and controllers. To simulate a real NCS, the plant and controller agents have to accomplish all of the relevant tasks performed by real plants and controllers. In principle, the plant agent samples data from the environment by accessing the plant status variables at a specified interval, writes the sampled data into a message packet, and sends it to the controller. When the controller receives the sampled packet, it generates a control signal $u$, and writes it into a packet that will be delivered through the network to the plant. The plant applies $u$ to the actuator as soon as it receives the control packet. The NS-2 simulator records each network event during the messages’ transmission through the network such as queuing at router buffers, dequeuing, and finally delivery to the controller. A clock-driven sensing and event-driven control-actuation approach is adopted in our simulations in accordance with real-time control systems used in industry.

As a discrete event-driven simulator, NS-2 assumes that a system can be represented by a set of state variables. The variables change value only at a countable number of points in time. The NS-2 simulator maintains a set of future events (such as message arrivals or timer firings) in a queue, and NS-2’s scheduler object dispatches and schedules events one at a time according to their simulation-time timestamp. Simulation time is calculated by NS-2 with a global virtual clock. During the simulation, the virtual clock may increase its value whenever a network event is triggered. NS-2 does not support scheduling of events in the past. Given this simulation environment, we therefore needed to consider how to schedule NCS ‘events’ in a network control scenario.

Fig. 3 shows simulation steps in a networked control scenario, assuming that in a sampling period the sampling and control packets are transmitted successfully (During the simulations, sampling or control packets may be lost under heavy traffic conditions, in which case actuation events will not be generated). In NS-2, a trace file can be generated for all network events scheduled and executed. It records the events and their respective simulation times in a predefined format. As shown in Fig. 3, every event is scheduled by the scheduler in time order. Synchronization events are triggered only at the time of the sampling and actuation events. The plant’s status is updated at synchronization.

The dotted arrow in Fig. 3 shows an infeasible sampling event which begins at the time of the arrow’s origin, and which should have been scheduled and executed by the time of its end point, but cannot be implemented because scheduling past network events is not allowed in NS-2. In other words,
synchronisation between the continuous plant simulation and the discrete network simulation can be achieved only if all sampling and actuation events are entered into NS-2’s event scheduler before their nominal time of occurrence (in simulated time). In our model all actuation events are scheduled by NS-2’s scheduler and periodic sampling events can be known in advance. However, some sampling events triggered by the continuous-time plant’s state may occur between two consecutive discrete-time simulation instants, so it may be too late to schedule them by the time they are detected by the simulation.

To understand the infeasible situation in Fig. 3 more clearly, the event scheduling algorithm for Fig. 3 is given in Fig. 4. Notations in Fig. 4 are explained as follows: \( F(\cdot) \) is the plant evolution function; \( t_k \) is the timestamp of the \( k \)th network event; \( t_N \) is the termination time; \( y_k \) is the plant state at time \( t_k \), and \( u_k \) is the plant input at time \( t_k \), respectively. The infeasible return is clearly shown in line 13 of the algorithm (Fig. 4). To deal with such ‘plant state events’ which are not schedulable, we will design and implement a special synchronisation mechanism, as will be described below in Section IV-A.

Algorithm: Event Scheduling in Plant Agent

Input: \( y_{k-1}, u_{k-1} \) or \( u_k \)
Output: \( y_k \)
1: Initialize \( k = 1 \); //event sequence number \( k \)
2: While discrete-time network event occurs at \( t_k \)
3:   If \( t_k \geq t_N \)
4:     Exit from the while loop; //go to line 18
5:   End If
6:   If the network event is a sampling event
7:     \( y_k = F(y_{k-1}, u_{k-1}, (t_k - t_{k-1})) \);
8:   Elseif the network event is an actuation event
9:     \( y_k = F(y_{k-1}, u_k, (t_k - t_{k-1})) \);
10:  End If
11:  If a sampling event triggered by the plant’s state (event state) is detected
12:    If its timestamp \( < t_k \)
13:      Return infeasible. //with failure
14:  End If
15:  End If
16:  Increment \( k \); //next event;
17:  End While
18:  Return with success.

Fig. 4. Event scheduling algorithm corresponding to Fig. 3.

IV. DESIGN AND IMPLEMENTATION OF THE HYBRID SIMULATION METHODOLOGY

Having considered the architecture and overall design of our simulator, here we describe specific technical features needed to make NS-2 suitable for simulating NCSs.

A. Synchronisation Model for the Simulation Environment

One of the most important challenges faced in our hybrid simulations is the need for synchronisation between the event-driven discrete network simulation, using NS-2’s networking components, and the continuous-time plant simulation based on numerical integration, using the ODE solver in plant agent objects. Correct synchronisation is a key issue that influences the accuracy and efficiency of the simulation significantly.

The events exchanged between the discrete-event network simulation and the continuous-time plant simulation are:

- The continuous-time plant simulation may send asynchronous ‘state events’ whose timing depends on the values of state variables. For example, when the measured difference between the plant’s state and its intended value exceeds a given threshold, the sampled data will be written into a packet and sent out immediately, to alert the controller.
- The discrete network simulation sends regular ‘actuation events’ to the continuous-time plant simulation to update the actuation signals.

State events travel from the continuous-time plant simulation (CPS) to the discrete-event network simulation (DNS), and actuation events go from the DNS to the CPS.

To synchronise the discrete-event network simulation and the continuous-time plant simulation in the simulator, we designed a synchronisation model based on predictable state events. Our synchronisation method borrows partly from a generic synchronisation model that respects the canonical algorithm [28], [29]. As for the discrete-event network simulator, NS-2 can easily manipulate every actuation event in time when its corresponding network event is triggered. However, the state events could not be manipulated in the same way. Instead, the discrete-event network simulation can shake hands with the continuous-time plant simulation at each discrete event time, and the state events generated asynchronously by the continuous-time plant simulation can be handled by the network simulation as soon as possible.

In the following, we explain the proposed synchronisation model and the prediction of state events.

The synchronisation model is shown in Fig. 5. The discrete network simulation and continuous plant simulation are assumed to be synchronised at time \( T_{\alpha} \), and the DNS executes all processes relevant to the current notified event (with zero time) and state events (with zero time). When an actuation event happens, the DNS sends the actuation event and its time stamp, \( T_{\alpha} \), to the CPS (arrow 2). Then the CPS will update the plant’s status at time \( T_{\beta} \) and predict the next occurrence of a state event. The updated plant status, the predicted state event and its time stamp, \( T_{\gamma} \), will be sent to the DNS (arrow 4). Similar to the network event, the predicted state event is also queued in NS-2’s event scheduler and executed according to its time stamp in the DNS. Like the synchronisation step caused by a network event (arrow 1) the synchronisation step caused by the predicted state event (arrow 5) follows the same process.

When the time \( T_{\delta} \) of the predicted state event is reached,
a synchronisation request is sent from the DNS to the CPS (arrow 6). Finally, the CPS will update the plant’s status at time $T_e$ and resynchronisation between the DNS and the CPS is achieved (arrow 8).

Importantly, this synchronisation model is based on the fact that the occurrence of state events is predictable, typically by extrapolating the plant’s behaviour based on its recent history. That is, the continuous-time plant simulation must provide, at each hybrid simulation step, the time stamp of its next state event and send this to the DNS before advancing. For example, in Fig. 5 the state event is predicted at time $T_b$ and sent to the DNS successfully before its occurrence time $T_c$ is reached. It is then detected and executed by the DNS.

Typically, the CPS simulates a continuous-time plant process by an appropriate mathematical model of how the plant’s state varies over time, in the absence of any changes to its environment. Therefore, a state event and its time stamp can be predicted by the CPS if no actuator events happen between the time at which the prediction is made and the time of the predicted state event. For example, the CPS can predict the state event and its time stamp $T_e$ accurately only at the time $T_b$ of the most recent actuation event. However, it may be impossible to do so accurately earlier, at say time $T_a$, because the prediction cannot take into account the change in the plant’s behaviour induced by the future update to the actuator signal which cannot be known in advance. Thus, to avoid this problem (the infeasible scheduling situation detected in Fig. 4), a mechanism for state event prediction was designed in which state events must be predicted at least at the time of every actuation event.

To implement the synchronisation model in NS-2, we designed a discrete-time event scheduling algorithm on the condition that all state events are predicted and inserted into NS-2’s event scheduler queue before their times of occurrence, as shown in Fig. 7. It is seen from the algorithm in Fig. 7 that it is now feasible to schedule all events if the state events are predicted successfully. $N_{state}$ is the number of predicted state events whose timestamps are in the current sampling period in Fig. 7.

Obviously, some of predicted state events will be unnecessary if their time stamps are after the next actuation event. To reduce the computation cost of the prediction of unnecessary state events, the time interval considered by the prediction is set to be as small as possible (provided that it is equal to or larger than the interval between the time of the prediction and the next actuation event). For this purpose, a cost-effective method is adopted by the mechanism for state events prediction, in which state events are predicted at the time of every sampling event. Therefore, the CPS does not have to consider those state events whose time stamps are after the next sampling event, as shown in Fig. 6. In addition, the method adopted leads us to achieve synchronisation at the time of every sampling event.

As a result, in consideration of the accuracy and efficiency of the prediction of state events, the CPS must predict state events at the time of every sampling and actuation event, and state events are predicted and scheduled only if their time stamps are before the next sampling event. Fig. 6 explains the mechanism and describes all of the events in time order, including periodic sampling, actuation, and predicted state events in a sampling period (between $T_a$ and $T_d$). It is worth noting that the case in Fig. 6 shows just one example of prediction of state events where the two predicted state events

---

**Algorithm: Event scheduling with prediction**

**Input:** $y_{k-1}$, $u_{k-1}$ or $u_k$  
**Output:** $y_k$

1: Initialize $k = 1$; //event sequence number $k$
2: While network or state event occurs at $t_k$
3: If $t_k \geq t_N$
4: Exit from the while loop; //go to line 24
5: End If
6: If the event is a sampling event;
7: Predict state event;
8: If $N_{state} > 0$ //a new state event detected
9: Update the old state event;
10: End If
11: $y_k = F(y_{k-1}, u_{k-1}, (t_k - t_{b-1}))$;
12: Elseif the event is an actuation event
13: predict state event
14: If $N_{state} > 0$
15: Update the old state event;
16: End If
17: $y_k = F(y_{k-1}, u_{k-1}, (t_k - t_{k-1}))$
18: Elseif the event is a state event
19: Schedule corresponding sampling event;
20: $y_k = F(y_{k-1}, u_{k-1}, (t_k - t_{k-1}))$;
21: End If
22: Increment $k$; //next event
23: End While
24: Return with success.
are valid to be scheduled (their time stamps \( T_c' \) and \( T_c'' \) are before the next sampling event). During the simulations, other situations are possible (e.g., no state events are valid in some sampling periods).

In our implementation of the prediction mechanism for state events, a predicted state event may be updated if an actuation event is scheduled before the time stamp associated with the predicted state event is reached in a sampling period. The update can be implemented in NS-2’s event scheduler because NS-2 is capable of cancelling an event from the event queue. For example, in the case shown in Fig. 6, a state event and its time stamp \( T_c' \) are predicted and sent to the DNS at the first sampling instant \( T_a \). However, an actuation event subsequently occurs at time \( T_b \), before the predicted state event’s time \( T_c' \) is reached. Therefore, a new prediction is made at time \( T_b \), along with a new time stamp \( T_c'' \), and the original state event is replaced in NS-2’s event scheduler by the new predicted event at the same time. In effect, the state event predicted for time \( T_c' \) is cancelled and replaced by an updated prediction for time \( T_c'' \). The DNS then performs the predicted state event at simulation time \( T_c'' \), corresponding to the time stamp \( T_c'' \) on the most recent state event prediction. Note that the example shown in Fig. 6 is a scenario with one update of the predicted state event. The update of predicted state events may happen again and again if several actuation events are scheduled in one sampling period during the simulation.

In the implementation of our state prediction methodology, the CPS computes the timestamp of the state event and sends the state event and its timestamp to the DNS if the timestamp is before the next sampling event. Therefore, the overhead of the state prediction depends on the number of states that are associated with predefined state events. For general control systems, the number of state events is limited and the overhead of state prediction is very small, even negligible.

B. Adjusting the Sampling Period

In our approach, the sampling period is defined as the time interval between two subsequent sampling events as shown in Fig. 6. Traditionally, sampling events have been periodic in control systems and the sampling period was assumed to be known in advance. Most simulation tools support only constant sampling periods. However, some co-design methods for networked control, such as the adaptive sampling period algorithm, require a simulation environment that supports reconfiguration of the sampling period during simulation [7], [19].

To satisfy this requirement, our approach used NS-2’s event scheduling mechanism to make the sampling period flexible: a sampling event is scheduled only after the previous sampling event was executed by the simulator. In Fig. 6, when the time \( T_a \) of a sampling event is reached, the sampling event at time \( T_a \) is executed and the next sampling event at time \( T_d \) is generated and queued in the simulator’s scheduler. In this way, consecutive sampling periods do not have to be of the same duration, and the period can be reconfigured dynamically during the simulation. In NS-2 this was done by programming the sampling period’s behaviour via a Tcl script.

C. Plant and Controller Dynamics

In the developed hybrid simulation environment, plant and controller agents are implemented by integration of ordinary differential equations and incorporation of control algorithms into NS-2, respectively.
This implementation requires four steps: modelling plant processes using mathematical formulations and writing these formulations in C++ code; programming control algorithms for controllers in C++ code; integrating these functions as methods of a derived agent class (PlantAgent and ControllerAgent); and building a link between the new C++ classes and variables and their counterparts in Tcl. The link makes Tcl understand functions and variables in the new C++ classes, so the plant and controller agent objects can be generated and associated with a node in Tcl scripts.

In our hybrid NCS simulation, the function of controller models is to compute actuation signals and send them to the plant (to apply to an actuator that affects the physical process) after receiving a sampling packet from the plant (containing values read from an environmental sensor). A plant agent is used to simulate continuous plant models such as a second-order plus delay process (SOPDP). As described in Section III-B, although the developed hybrid simulation environment only provides a few pre-defined plant process models, new models can be easily programmed and added as C++ subroutines by use of existing continuous system simulation techniques.

V. Case Studies

To illustrate the capabilities of our hybrid simulation environment developed for computer control systems over data networks, this section presents case studies to show how to model quality-of-service (QoS) adaptive control in a simple network scenario with a dumbbell structure using our approach. As a control methodology, QoS-adaptive control is receiving significant attention in the networked control literature [13], [5]. Note that the case studies focus on hybrid system modelling and simulation, rather than on control strategy design and system stability analysis.

Generally speaking, a controller has only one setting of controller parameters such as constant integral gain. However, constant controller parameters are generally pre-defined without consideration of varying message transmission delays such as the delay experienced by the sampled packet between a sensor and an actuator. The main idea behind the QoS-adaptive control method is to adapt controller parameters (e.g., controller gains) in response to current network traffic. The QoS-adaptive controller has multiple controller settings associated with different messages transmission delays. It is also adopted by other control methodologies in NCS research such as multi-rate proportional-integral-derivative (PID) controllers, end-user control adaptation, and queuing communication architectures for NCSs [5], [4]. In this case study, the sensor and the controller are assumed to be able to measure the message propagation delay as the QoS metric.

A part of the Tcl script for the simulations is shown in Fig. 8. A plant agent and a controller agent are built and the two agents are attached to two nodes (node(0) and node(5)) individually. Value sample_period is the initial value of the sampling period, which is set to 0.03 second. A trace file is set to record the plant’s state that is measured during the simulation.

\[
\begin{align*}
\text{set plant(0)} & \text{ [New Agent/PlantAgent]} \\
\text{set controller(0)} & \text{ [New Agent/ControllerAgent]} \\
\text{$\text{ns attach-agent}$ $\text{Snode(0)}$ $\text{Splant(0)}$} \\
\text{$\text{ns attach-agent}$ $\text{Snode(5)}$ $\text{Scontroller(0)}$} \\
\text{$\text{ns connect}$ $\text{Scontroller(0)}$ $\text{Splant(0)}$} \\
\text{set sample_period 0.03}
\end{align*}
\]

Fig. 8. Part of the front-end TCL script for the case study.

A. Assumed Communications Scenario

The network is assumed to contain eight nodes communicating over a wired network, including plant and controller nodes, switches and four nodes for simulating traffic congestion. As usual, the plant node is a combination of a sensor and an actuator. Fig. 9 shows the logical network architecture, which is also a simulation snapshot. All communications links are configured to have a queue of length 50 and the queuing algorithm is Drop Tail. The propagation delays are 20 ms for the link between the two switches and 10 ms for all others links. The bandwidth of every link is 1 Mbps.

![Communication scenario](image)

Fig. 9. Communication scenario.

It is assumed that the sampling period is 30 ms. Within this sampling period, the plant node is assumed to read values from its sensor, and send them in a sampling packet of 100 bytes to the controller. After the controller receives the sampling packet, actuator signal values are computed by the controller and sent to the plant node, for application to the actuator. The size of the control packet is also 100 bytes. All traffic is constant-bit-rate. The traffic loads caused by message transmission for the control system and simulating traffic congestion are about 53.12 kbps and 1029.33 kbps, respectively.

B. Control System Models

Consider a second-order plant process described by the following ODE equation, which represents a DC electric motor drive [4],

\[
\begin{align*}
\ddot{y}(t) - 28.586\dot{y}(t) - 60.36184y(t) + 2029.826u(t) &= 0, \\
y(t_0) &= y_0, \quad u(t_0) = u_0
\end{align*}
\]

(1)
where $y(t)$ is the plant output; $u(t)$ is the plant input, i.e., controller output; and $y(t_{i})$ and $u(t_{i})$ are their initial values that can be configured in the Tcl script for constructing the simulations. The discrete-time algorithm adopted in the simulations of the plant dynamics is a fourth-order adaptive Runge-Kutta method.

The algorithm for the proportional-integral (PI) controller is as follows,

$$u(t) = K_c \cdot e(t) + \int \frac{K_c}{T_i} \cdot e(t) dt, \quad e(t) = y_r(t) - y(t)$$

(2)

where the proportional gain $K_c$ and integral time $T_i$ are tuned as $K_c = 0.1701$ and $T_i = 0.45$, respectively. $y_r(t)$ is the set-point of $y(t)$, and $e(t)$ is the control error. Thus, the discrete-time single-step algorithm for the PI controller can be obtained,

$$u_m = u_{m-1} + K_c(e_m - e_{m-1}) + \frac{K_c T_m}{2 T_i}(e_m + e_{m-1})$$

(3)

where $u_m$ and $u_{m-1}$ are the plant inputs (i.e., controller outputs) at the time of the $m$th and $(m-1)$th sampling periods, respectively; and $e_m$ and $e_{m-1}$ are the measured error at the time of the $m$th and $(m-1)$th sampling periods. The duration of $m$th sampling period $T_m$ is set to 30 ms. For simplicity, the time cost of the computation in the controllers is set to 0. Equation 3 tells us how to implement the discrete control algorithm and compute the plant input by the controller in the $m$th period.

Gabel designed a QoS-adaptive controller with controller parameters associated with the transmission delay $\tau$ between the sensor and the controller for the sampled packets [13]. As a co-design methodology, a QoS-adaptive controller adapts its control parameters based on the network’s QoS parameter that is measured online. The discrete control algorithm can thus be improved as follows,

$$u_m = u_{m-1} + K_c(\tau_m) \cdot (e_m - e_{m-1})$$

$$+ \frac{K_c(\tau_m) T_m}{2 T_i(\tau_m)} \cdot (e_m + e_{m-1})$$

(4)

where $K_c(\tau_m)$ and $T_i(\tau_m)$ represent improved controller parameters based on the measured transmission delay $\tau_m$ at the $m$th sampling period. The equation shows that networked induced delays and packet losses are considered partially. Based on $\tau_m$ the controller’s parameters for $K_c(\tau_m)$ and $T_i(\tau_m)$ are set as per Table I. The parameters with no delays as the original ones are also listed. Also note that Equation 4 allows for the possibility that the sampling interval $T_m$ is different from one period to the next although it is set to be constant during the simulations. It is necessary for our simulation environment when some co-design methods like the adaptive sampling period method are evaluated.

C. Simulation Results

By using our hybrid simulation environment, the results with QoS-adaptive control and a base line result with PID control are shown in Fig. 10. The example on the left has no traffic congestion (total traffic load is 53.12 kbps), and the example on the right has traffic congestion (total traffic load is 1082.453 kbps). In these simulations, the desired DC motor speed (the plant reference) is 50 rad/s and the sampling period is set to 30 ms. During the simulation, every network event is logged. It can also be displayed visually by use of the Network Animator (NAM) for NS-2 as shown in Fig. 9.

In both cases shown in Fig. 10 it is seen that the second simulation (right plot of Fig. 10) reveals the influence of network congestion on the controllers’ behaviours. Also it demonstrates that the QoS-adaptive controller produces a more stable output than the PID controller, confirming the advantage of the adaptive control in a congestion situation.

To the best of our knowledge, existing simulators for NCS simulation do not consider any specific network properties (e.g., network architecture and link capacity). For example, the simulation in Gabel’s work [13] is based on an assumption that the probability density function of the delays is uniformly distributed between a limited region. Compared to these simulation methods the hybrid NCS simulation environment developed in this paper considers communication networks more explicitly, and makes it possible to evaluate co-design methodologies such as QoS-adaptive control in a specific network scenario.

VI. CONCLUSION AND FUTURE WORK

This paper has described a hybrid system simulation environment, which is an NS-2 based simulator for networked control systems. The approach has focused on co-simulations in which discrete-event communication networks are simulated in parallel with continuous-time modelling of physical plant processes. The objective is to provide an effective and easy-to-maintain way of simulating continuous-time systems dynamics in a discrete-event networking environment.

To achieve this we implemented a continuous plant process simulation algorithm within NS-2’s discrete networking

| Table I: Controller settings for different delays [13]. |
|---|---|---|---|---|---|
| Delays (ms) | 0 | 15 | 30 | 45 | 60 |
| $K_c$ | 0.1701 | 0.1584 | 0.1180 | 0.0854 | 0.0768 |
| $T_i$ | 0.4500 | 0.4439 | 0.4816 | 0.4509 | 0.4646 |

![Fig. 10. Plant status measurements with the QoS-adaptive control and PID control (with original parameters).](image-url)
model. A special synchronisation mechanism was devised for integrating asynchronous state-triggered events into a discrete time-driven simulation. A mechanism was also introduced to allow the control system’s sampling period to be varied dynamically. The resulting software takes both digital network characteristics and feedback-driven control system performance into account. Furthermore, the simulation environment takes advantage of NS-2’s large number of ready-to-use network protocol models. However, the number of pre-defined plant models for continuous-time system simulations is yet to be extended, and needs to be emphasised in the future.

ACKNOWLEDGEMENTS

This work was supported in part by the Australian Research Council under Discovery Projects grant number DP0773012 to C. Fidge, by the Australian Government’s Department of Education, Science and Training under International Science Linkages grant number CH070083 to Y.-C. Tian, and by the Natural Science Foundation of China under International Collaboration grant number 60774060 to Y.-C. Tian.

REFERENCES