A Modular and Layered Cosimulator for Networked Control Systems

Stefano Falasca†, Christian Belsito†, Andrea Quagli* and Antonio Bicchi*

***Abstract*— In this paper we present a simulation framework that is intended to be used for a wide class of network control systems. It is designed in order to allow interaction with the Matlab/Simulink environment. The low-level structure is written using the C++ language so that hardware/network in the loop simulation can be readily done by substituting an arbitrary set of software components with corresponding hardware in an easy way. Sharp modularity of the code structure permits to adapt the simulation to different needs changing only a few modules.**

We shall show the results of its specialization to a recently proposed controlling approach, demonstrating the flexibility and the accuracy of the simulator.

***Index Terms*— Networked systems, Modelling and simulation, Industrial automation, manufacturing.**

I. Introduction

Network control system (NCS) is a relatively new research field. It has proven to be highly stimulating for researchers with different backgrounds. Much effort has been invested in researching this interesting field and several promising approaches have been proposed (and used in order to develop useful results) [9], [2], [3], [5], [6], [11], [12], [15], [16].

Research in this field is mainly based on the development of purely analytical results. These results are usually found after developing a new model for the NCS. Statements like ”this appears to be a novel formulation” and ”by such modelling it is possible to describe most of the paradigms proposed so far” are quite easily found in NCS related papers. Although it is very easy to find papers (even recent ones) containing no simulations at all, various authors, in recent years, have simulated simple systems in their papers. Particular emphasis has often been put on the description of the implementation of the used simulator.

Significant efforts have been recently devoted to the development of NCS simulators (e.g. [7]). So far, however, simulators only work with the same network control paradigms proposed by their own authors.

These are the reasons why we have developed a simulation framework aimed at being highly modular, able to provide support to a variety of existent and new approaches and – what seems to be new in this area – suitable for network in the loop (NIL) as well as hardware in the loop (HIL) simulations. The latter requirement allows to ensure coherency between simulation code and the code used during HIL/NIL simulations.

This work was supported by the EC under contract ”CHAT - Control of Heterogeneus Automation Systems”

*Authors are with the Interdept. Research Center “Enrico Piaggio”, University of Pisa, via Diotisalvi, 2, 56100 Pisa, Italy. Phone: +39050553639. Fax:+39050550650. andrea.quagli@centropiaggio.unipi.it, bicchi@centropiaggio.unipi.it

†Authors are postgraduate students at the Faculty of Engineering, University of Pisa. stefano.falasca@gmail.com, christian.belsito@gmail.com

As it has been pointed out (e.g. in [8]), it is highly convenient to have the ability to simulate the NCS right into the Matlab/Simulink environment so that one can easily take advantage of all the control design tools available there. Nonetheless it appears to be appropriate to give a ”low level” implementation of the components constituting the simulator in order to promote code reuse. For this reason every single component is implemented in C++ (the implementation being completely unaware of the Matlab/Simulink environment) and wrappers are written for Matlab.

As it has been said before, several different approaches to the control over network problem have been proposed so far (and many will come in the future). It will be useful to distinguish the control platform and the control strategy. Describing a network control platform means defining the topology of network connections as well as the network capabilities of transmitting data and guaranteeing data consistency. Choosing a control strategy means to define how received data are used to produce a control law and how to tackle typical NCS’s problems such as packet dropouts (if such behaviour is possible within the given control platform) and, of course, transmission delays. As far as we know this distinction has never been pointed out before. Our belief is that this is a possible source of misleading model formulations.

This paper has the following structure. In section II it is possible to find a description of the proposed simulation environment. Section III points out the control platform and control strategy proposed in [4] and describes their casting into the simulation environment’s structure. Section IV presents the results of two different simulations, the first one (section IV-A) being related to a very simple example and the second one (section IV-B) being a complete robotic arm. Conclusions follow.

II. Simulation Environment

The purpose of this section is to describe the implemented simulation environment in order to allow the reader to use it for his objectives. The main emphasis is on giving a general picture of the class of NCSs for which this simulator is intended to be use. It is worth noting that not every NCS problem studied so far is included in this class. The reader is encouraged to read one of the available NCS’s surveys (e.g. [14]) where problems such as adaptive remote control, networked based auto-tuning, peer-to-peer NCSs are cited and described.

It is convenient to recall and to be more precise about what the distinction between control platform (CP) and control strategy (CS) is about. The CP specifies the configuration of the NCS and the kind of network at one’s disposal. The CS specifies how the control problem is tackled and

how data communication capabilities are exploited. Reader is invited to consider this simple example where every bit of information used to describe the scenario is followed by a tag contextualizing it. A 2-input/2-output (CS) plant is connected with a controller by means of a wireless interface (CP). Each smart sensor providing output has its own wifi interface (CP). The actuators are connected to an embedded system that generates the commands by using data received from its network interface (CP & CS) and environmental data that are locally sensed (CS). A controller computer is located at distance such that only a portion of the data sent by the sensors reach it (CP); in order to overcome this problem the controller extrapolates the output behaviour when needed (CS). Controller's computations lead to the generation of a control law that is sent over the network (CS). When the plant does not receive control packets for too long it enters a safety mode (CS).

CP is what the proposed simulator is mainly about. The proposed CP defines whether a given NCS is liable to be simulated within this environment or not. CS, on the other hand, has not been defined in implementing this simulator; much attention has been paid to make it possible to use a variety of CS in this context. Proposed CSs have been, of course, taken into account when defining the simulation framework; the authors hope that it will be possible to use the resulting implementation for CSs to come.

The possibility of HIL simulations permeates every component of the simulator; it is therefore described through the whole section.

A. Modularity and Stratification

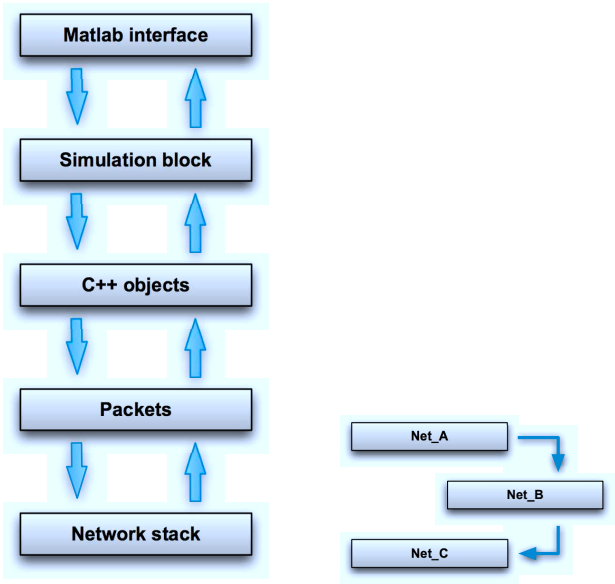
In deciding how to implement the simulation infrastructure some criteria had to be met:

1) independence from the external environment (e.g. Matlab/Simulink);
2) encouragement of code reuse in terms of the actual implementation of a real network controlled system (code reusing allows fast and less error-prone implementations);
3) the possibility of doing HIL/NIL tests by substituting an arbitrary set of software components with corresponding hardware in an easy possible way;
4) low effort extension of the given code (if needed) and
5) ease in connecting the simulator to existing network simulation infrastructures.

In order to reach such goals we decided to introduce a stratification for every single component, as well as a modular division.

As a first step toward stratification the simulator implementation has been detached from the Simulink environment. This allows an easy porting of the overall structure toward new simulation environments as well as (possibly several) embedded computer systems. Every single Simulink block uses an underlying C++ object in order to fulfil its duties; by this choice no relevant Matlab code is executed at any time during the simulation. Attention has been paid to making every component capable of fully asynchronous operations so that executing every piece of code on a different processor will not be an issue. This allows the substitution of a Simulink block with the desired hardware without knowing its actual implementation.

The second step made toward stratification is the designing of *smart packets*. This will be clarified in section II-C. The resulting software stratification structure is represented in figure 1(a).



(a) (b)

Fig. 1. Representation of software stratification and communication channel modularity

The whole simulation framework can be seen as composed of smaller subsystems. Modularity is achieved by encapsulating every simulator entity working independently and asynchronously from the others. An example of such a sharp modularity referring to the communication channel can be seen in figure 1(b). Details on the actual module structure chosen can be found in section II-B.

B. Control Platform

The components that constitute the NCS are: the plant, the controller and two network channels which allow data to be sent from the controller to the plant and vice-versa. Separating the network into two different channels allows the user of the simulator to easily model asymmetric communication conditions (it is, of course, possible to join the two channels if needed). Most of the time, if the network separation capability of the proposed simulator is used, the two channels will probably be unidirectional. Figure 2 shows the described structure.

All network related components make use of a single wrapper class (NetSocket) that hides all the necessary operating system calls and delivers the functionality needed for building up and managing a unidirectional or bidirectional communication channel. This allows a complete decoupling from the operating system in use and therefore the switch to a different platform (e.g. embedded real time operating

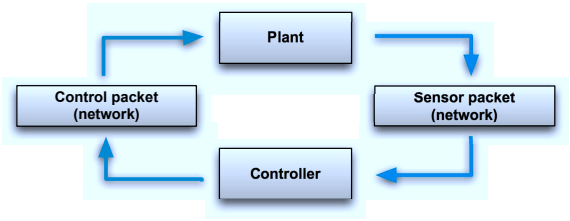


Fig. 2. Control platform.

systems) would simply require the reimplementation of the relative routines of NetSocket. This architecture provides also the ability to use simulated networks, real networks (NIL) or a combination of both. As we will see the actual implementation of NetSocket used for the simulations presented later in this paper makes use of the TCP protocol. To use protocols other than TCP it would be sufficient to reimplement only this software component leaving the others untouched. To make NetSocket more powerful and to allow its use in a broader range of scenarios (e.g. NIL or using it with existing network simulation programmes) the operating mode can be server (i.e. accepting incoming connection requests) or client (i.e. connecting to listening server ports). This can be useful when modifying components others than the simulator is not possible for some reasons. NetSocket's interface is packet oriented and completely unaware of the packet's data format.

For the reasons stated before the simulator makes use of unidirectional data streams. Each stream is composed of 3 main blocks:

- sender block (Net_A)
- communication channel (Net_B)
- receiver block (Net_C)

This simple yet powerful structure is represented in figure 1(b). This design choice allows a simple communication channel replacement (e.g. with a real communication channel or a simulated one). This leads to the ability of fast and easy robustness analysis with respect to:

- the use of a different network stack;
- the introduction of limited variable delays;
- packet-loss or data corruption and
- presence of multiple agents on the network.

Therefore it is possible to fine-tune the behaviour of the system before deploying it in a production environment. Besides this, such a structure provides the ability to obtain statistics about the network channel usage and behaviour like average and min/max packet throughput, informations regarding non deterministic delays introduced by the network stack (most of the stacks available on today's mainstream operating systems introduce unknown delays) and so on.

By virtue of the described structure it is also possible to get rid of the network stack and easily substitute it with a simple data-passing model such as the one described in [14] or with complex networks like the ones in [10], [13], [1]. Simple channel models are used very often in defining the control platform. In such cases using a proper network stack would probably be the source of misleading results. Nonetheless it

```
class Controller{
  private:
    //...
  public:
    Controller();

    void dataIn(char* updateSerial,
      double timestamp,
      double* state,
      double* cmd);
    void exec();
    void dataOut(double*& startState,
      double* startTime,
      double* resetPort,
      char*& toTcpSerial);

    ~Controller();
};
```

Listing 1. A possible controller automaton interface

can be useful to test such control platforms in a more realistic environment; this simulator allows to plug in different kinds of network with no effort.

An application simulating the network with its delays is also presented. It is structured as a multi-threaded application and uses the Posix API to provide real time timing control.

C. Simulator's Capabilities in Describing Control Strategies

Control strategy, as has been said before, is not defined within the implemented simulation environment. This is because of the will to make the simulator able to be used as far as different CSs are concerned. Flexibility of the proposed design is the main concern of this section.

Our main concern has been writing a simulator which could be easily modified with minimal prior knowledge about it. In such a context the purpose and algorithms of functional blocks vary often. The proposed implementation provides a common interface for every functional block.

The basic idea behind the design of every single functional block is that they have to behave like automata, thus having internal states, input vectors and output vectors. Input data as well as output data can be of different nature. A functional block implementing the network protocol, for example, can have the set of sensors' acquisitions as its input and a packet to be sent over the network as its output. The controller block, on the contrary will have a packet as its input data as well as its output. The automaton's state is intended to represent its memory and can be used to determine which step of the algorithm it is into as well as some kind of history of received inputs. Listing 1 is part of the interface of the controller class used for the simulations presented later in this paper.

By such representation of functional blocks asynchronous operations are made possible. It is possible, for instance, to have an automaton whose goal is to receive packets from the network and to send appropriate commands to the plant's actuators (in a HIL scenario this automaton will execute on an embedded processor). On the other hand these packets would have been sent by another automaton executing right into the Matlab/Simulink environment simulating the controller behaviour. It is clear that such a

scenario would lead to arbitrary operation interleaving. The behaviour of the described system would be as follows: the receiver's embedded computer would execute a thread that, through polling, senses the network waiting for a new packet. Thread's operations would be simply the calling of the exec() automaton's function followed by the calling of the getData() function. The results of getData() would be, at each execution, the best command to be given to each actuator.

The coding paradigm described so far has been used in implementing Net_A, Net_C and Net_Socket objects and needs to be used when implementing the application specific controller block.

In order to complete the description of what makes the proposed simulation environment able to be adapted to a variety of CSs the packet structure has to be described.

Different control platforms and control strategies often perform similar tasks in different portions of the control loop. We shall consider as an example a control platform having only a couple of sensors and a controller algorithm that allows the refresh of only one state at a time. The sensors' data are very likely to be sent on a single packet and the controller computer will execute the network protocol in order to choose between the two states. If, on the contrary, a lot of sensors are spread on a large plant, each one having a separate network access they must execute a network protocol in order to decide which one will gain the network access. In order to tackle this problem the authors' decision has been to *empower* packets by giving them, beyond the mere task of containing the payload, also data-managing functionalities. Packets, in fact, represent the entity travelling over the network and, therefore, can be seen as something similar to shared data structures. Packets' member functions, therefore, can execute on the plant's computer as well as on the controller's computer, allowing the user to adapt the simulation environment to its needs. Moreover packets provide the data-carrying entity and, therefore, it is actually a good idea to let their implementation, depending on the type of the packet, be the only piece of code aware of what needs to be communicated over the network and, of course, how data are encoded.

In order to clarify what has been said so far it is appropriate to consider the scenarios presented above in more detail. For the sake of ease of discussion only the case in which every single NCS's component is simulated will be considered hereafter. For the sake of simplicity only the fully-simulated case is considered hereafter.

1) The full plant state is transmitted over the network at each network access. This can be done by instantiating a packet object having a public method that has the timestamp and full state information as input and that builds a data structure containing them. This object can then be serialized and sent over the network.
   The controller, on the other side, has to execute a network protocol in order to build an estimate of the plant's state. This can be easily done by building a packet object by de-serializing the received data. Such an object should have a public method that, given a state estimate, uses one of the sensors' value received in order to enhance the estimate.
2) Only one plant state (coming from a single sensor) is transmitted over the network. This can be implemented through the use of a packet object having a public member which, given sensors' data, executes a network protocol and builds a data structure containing the value read by the appropriate sensor. This object can then be serialized and sent over the network.
   The controller has to use the received data to build an as good an estimate as possible. It is possible to do so by building a packet object (by, as usual, de-serializing the received data) and, therefore, calling a public method of such an object that, given an estimate of the plant's state, uses received data to improve it.

```
class Packet_StateUpdate : public Packet_Interface{
  private:
    // packet data goes here
  public:
    //...
    void createSerial(char*& serial);
    void loadSerial(const char* serial);

    double* buildEstimate(double* estimate);

    void dataSet(double* externalState);
};
```

Listing 2. Packet_StateUpdate interface

It is worth noticing that the solution described above is by no means the most straightforward neither to the first problem nor to the second. The reader, however, may have already noticed that the described data structures have common interfaces (that can be seen in listing 2).

The tipping point here is that simulating the two scenarios involves (by virtue of the common packet interface) the writing of two packet object's implementations and that the remaining simulation code would be the same.

As can be easily noticed there is no point in letting entities different from the packet itself knowing anything about how data are sent and how data are used as far as the proposed implementation is concerned.

The same reasoning led to the definition of the interface for the second type of packets that the NCS will use. This packet is intended to carry control laws as well as, within certain CSs (e.g. model-predictive based CSs) state estimates or tuning values. The interface can be seen in listing 3. Here, beside the serialization-related functions, class members simply allow to get and set the data that have to be carried by the packet.

Finally it is important to mention that the simulators' simulink structure uses a defined communication interface that allows a simple substitution with custom plant models. Insofar as the actual hardware plant is used in an HIL scenario this clarification is not needed. The block containing the system model (if any) must have the following input/output signals:

1) input vector u representing the commands sent to the model;

```
class Packet_Command : public Packet_Interface{
  private:
    // packet data goes here
  public:
    //...
    void createSerial(char*& serial);
    void loadSerial(const char* serial);

    double* getCmd(unsigned int step);
    void setCmd(unsigned int step, double* vect);

    double* getState(unsigned int step);
    void setState(unsigned int step, double* vect);
};
```

Listing 3. Packet_Command interface

2) output vector y representing the model output values;

some particular CSs may require it to provide the system with additional signals (e.g. output vector x representing the state variables; paremeters' values; ...).

III. Delay Compensation in Packet-Switching Network Control Systems

In this section the control platform and control strategy proposed in [4] are briefly reviewed.

Afterwards a description of a specialized version of the simulation framework applied to this particular case is given.

Interested readers are invited to note the ease with which the hybrid formulation proposed in [4] has been translated into an equivalent problem that can be simulated within the simulation framework.

A. Problem Description

In [4] the authors consider the problem of stabilizing sufficiently smooth non-linear time-invariant plants over a network where feedback is closed through a limited bandwidth digital channel. Reliable packet-based networks are explicitly considered, for which both the time between consecutive accesses to the network and the delay by which each data packet is received, processed, and fed back to the plant are unknown but bounded. A model-prediction based strategy is used.

One of the main problems in the NCSs is how to perform the control action minimising the network congestion. For this purpose the relatively high payload which can be associated to each packet is exploited. Whenever the data are small enough to be encoded in a single packet, the associated communication overhead remains the same. It is therefore a profitable choice to take advantage of this property and send, whenever possible, packets of the maximum allowed size.

For such reason the controller's computer provides the plant with a *long* feed-forward control signal, valid between two consecutive transmission instants. Roughly speaking, at each reception of a new measurement the controller updates an internal model-based estimate of the current state of the plant. Based on this estimate, the controller computes a prediction of the control signal on a fixed time horizon by simulating the plant's behaviour. This signal is then coded and sent in a single packet during the next network access. When received by the plant it is decoded and resynchronized by an embedded computer, this action is performed according to the time-stamping of the original measurement.

A detailed description of the platform and the control strategy is provided in the following sections.

B. The Control Platform

The digital network is seen as a couple of reliable unidirectional communication channels. The first one's task is transmitting control signals (i.e., it connects the controller to the plant). The second one (the one from the plant to the controller) is in charge of transmitting (partial or total) state measurements. On each channel only one node can send its information at a given time (i.e., if several sensors are present, only partial instantaneous knowledge of the plant's state can be achieved). Of course this is not an issue as far as the control side of the network is concerned (the controller is the only node sending data over this channel).

The presence of a network protocol is required. This protocol is an algorithm that chooses at each time instant which node is granted the access to the communication channel. Such decision is based on the error z (being the difference between an estimate of the plant's state and its actual state) and, of course, on data availability. A protocol is said to be UGES when it imposes an exponential convergence of the discrete update law induced by the protocol. The definition of invariably UGES also requires that this property remains valid when the update is not made at each step but according to an arbitrary increasing sequence. Using an invariably UGES protocol is a requirement within this context. There is a great variety of network protocols in literature. For example, a simple protocol is Round Robin (RR) in which the state sensed by node i is transmitted periodically with period N, where N is the total number of nodes. An example of a more complex protocol is given by the Try-Once-Discard (TOD) approach: access to the network is granted to the node with the greatest weighted error from the last reported value. For further details on such protocols, their classifications and for important results within this framework the reader is referred to [9].

The network is supposed to introduce variable delays. We assume that the time between two consecutive successful accesses to the network – the usual name given to this number is *maximum allowable time interval* (MATI) – is bounded both on the measurement side and on the control side. In the same way, we assume that the transmission delays, on both sides, cannot exceed a certain limit.

C. The Control Strategy

It is important to recall, as mentioned in the previous section, that the network channels only allow one node at a time to send their data. This constraint can be seen as dictated by the following reasoning: it is realistic to assume that the plant sensors are dislocated in different places (big chemical plants are easy examples of such an environment) thus making it impossible to have an instantaneous global knowledge of the latter. This can be taken account of by modelling an ad-hoc network protocol. Of course such a protocol has to be invariably UGES.

We recall that the approach consists in exploiting the possibly large payload available on each packet by sending, whenever possible, not only the value of the control law to be applied at a given instant, but also a prediction of the control law that will be applied in the future instants, obtained based on the (imprecise) model of the plant. The so-obtained control-packet, thus containing a sequence of control values valid on a given time-horizon, is then sent over the channel. The plant is now able to compensate the effects introduced by communication delays in the control loop via local re-synchronization. Details of this algorithm follow.

The plant computer is required to accomplish two main tasks. The simpler one is receiving the packets sent by the controller and deliver controls to the actual plant with the appropriate timing (we will refer to this as the *interpolating function*). The second one is choosing what portion of the available sensor data have to be sent over the network in order to allow the controller to generate a new control packet. This decision can be made according to several different criteria. (as described in section III-B).

The controller's computer uses received sensor data, state estimates (if any) and a (possibly inaccurate) plant model in order to generate two main data streams:

- a control law that is valid during a given time interval and
- the estimated behaviour that the plant would exhibit if controlled with the aforementioned control law (and if the model were correct).

Algorithm 1 *Plant computer* dealing with sensor data

```
loop
  Packet_StateUpdate = executeProtocol()
  gainNetworkAccess(sensor signal network)
  sendData(sensor signal network, Packet_StateUpdate)
end loop
```

Algorithm 1 shows a pseudo-code version of what the plant computer executes when dealing with sensor data. Operations are repeated indefinitely and the time between subsequent executions is variable.

The controller acts in the following way:

- at time t_j uses the data received from the sensor(s) in order to initialize the plant model state variables;
- starts the simulator and records its behaviour and the control values used;
- uses (a portion of the) recorded data in order to build a packet that will be sent to the plant via the network channel.

State variables are initialized by completing, if needed, the data received from the sensors by using the state estimate previously computed. Operations are, once again, indefinitely repeated and the time between subsequent executions is limited. At the start of the simulation, if more than one packet has been sent by the plant's computer, only the latest one will be used. Algorithm 2 shows a pseudo-code of what has been described so far.

Algorithm 2 *Controller* algorithm

```
loop
  repeat
    Packet_StateUpate = recvData(sensor signal network)
  until isNew(Packet_StateUpdate) = true
  τ_j = t
  initSimulator(Packet_StateUpdate, oldSimulationData)
  recordedData = 0
  repeat
    recordedData += getSimulatorOutput()
  until t − τ_j ≥ T̂_0
  Packet_CommandSequence=buildPacket(recordedData)
  gainNetworkAccess(control signal network)
  sendData(control signal network,
  Packet_CommandSequence)
end loop
```

Each packet created by the controller has its own timestamp assigned. This information is sent over the network and will be used by the plant's computer in order to synchronize with the absolute clock the predicted state variable values and commands to be used to control the plant.

In conclusion this strategy preserves global exponential stability under network communication, provided that the involved MATI and delays remain below a certain limit. Upper bounds on the previous quantities are explicitly provided, based on the parameters characterizing the quality of the plant's model and the stability properties of the protocol and the nominal closed-loop plant.

IV. Simulation Results

A. Single Integrator

In this example we show how to apply the simulation framework to an extremely simple dynamic system consisting of a single state variable x and one control input u.

The dynamics of the system are described by the following differential equation:

$$\dot{x} = x + u. \tag{1}$$

A suitable feedback control law that stabilizes the system starting from any initial condition is given by $k(x) = -2x$.

It is easy to see that this system, together with the given control law, does satisfy all the assumptions of [4]. This means that this system will exhibit a stable behaviour when controlled via the proposed CP/CS.

Figure 3 shows the results of the simulation of the NCS as well as the behaviour resulting by the use of a TCP stack as a communication channel.

B. Robotic Manipulator

To show how the simulation framework behaves in a more realistic environment a PRP[1] robotic manipulator described by the following generic differential equation was chosen:

$$B(\vec{q})\ddot{\vec{q}} + C(\vec{q},\dot{\vec{q}})\dot{\vec{q}} + G(\vec{q}) = \vec{\tau} \tag{2}$$

[1]The manipulator is constituted by a chain of three links connected via a prismatic joint, a revolute coupling and a prismatic joint

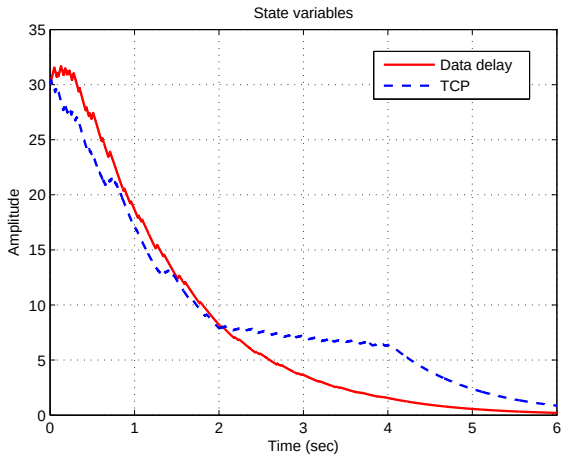


Fig. 3. Single integrator simulations - network delay: 0.01s.

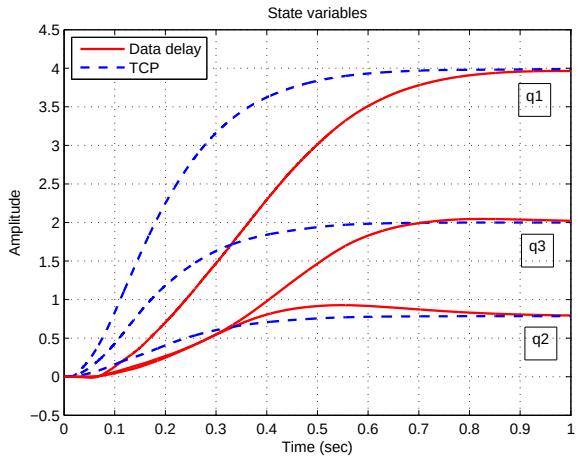


Fig. 4. Robotic manipulator simulations - network delay: 0.01s

For the sake of compactness the expansion of each term was omitted. The feedback control law chosen to stabilize the manipulator's end effector when a positioning task is considered is known as *Computed Torque*. The resulting controlled system is linear and stable but, nonetheless, it does not verify all the requirements of [4]. It is worth noting that, being the main result of [4] a sufficient (and not necessary) condition for the stability of the NCS, it is not meaningless to simulate it.

Simulations have been carried out for this system using the CP and CS described in section III. Moreover a TCP stack has been used as a replacement for the simple network model given in [4]. Figure 4 shows the differences between the two behaviours obtained. As it is possible to see both simulations exhibit a stable behaviour for the system.

V. Conclusions

In this paper the importance of having at one's disposal a cosimulator when researching the network control systems' field is pointed out. Such cosimulator should allow its user to substitute arbitrary portions of software by hardware components as well as third-party components' simulators.

The problem of writing a simulation framework being as flexible as the variety of different network control approaches proposed so far demands has been faced.

A strongly layered and sharply modular software solution is proposed.

We specialized the simulation framework to a recently proposed, complex network control paradigm in order to show the capabilities of the proposed framework.

The cosimulator can serve as a push toward the development of simulation modules implementing different network control paradigms.

References

[1] Anon. Scale communicates via profinet. In *Prof. Eng.*, volume 17, pages 45–46. 2004.

[2] R.W. Brockett and D. Liberzon. Quantized feedback stabilization of linear systems. *Automatic Control, IEEE Transactions on*, 45(7):1279–1289, Jul 2000.

[3] W. Brockett. Minimum attention control. In *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, volume 3, pages 2628–2632 vol.3, Dec 1997.

[4] A. Chaillet and A. Bicchi. Delay compensation in packet-switching networked controlled systems. In *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, pages 3620–3625, Dec. 2008.

[5] N. Elia and S.K. Mitter. Stabilization of linear systems with limited information. *Automatic Control, IEEE Transactions on*, 46(9):1384–1400, Sep 2001.

[6] H. Ishii and B.A. Francis. Stabilizing a linear system by switching control with dwell time. *Automatic Control, IEEE Transactions on*, 47(12):1962–1973, Dec 2002.

[7] M. Tabbara J.J.C. van Schendel, D. Nesic and W.P.M.H. Heemels. Networked control system simulation & analysis. Internal Report (2008) DCT 2008.119, Eindhoven University of Technology, Department of Mechanical Engineering, 2008.

[8] G.P. Liu, D. Rees, and S.C. Chai. Design and practical implementation of networked predictive control systems. In *Networking, Sensing and Control, 2005. Proceedings. 2005 IEEE*, pages 336–341, March 2005.

[9] D. Nesic and A.R. Teel. Input-output stability properties of networked control systems. *Automatic Control, IEEE Transactions on*, 49(10):1650–1667, Oct. 2004.

[10] E. Tovar and F. Vasques. Real-time fieldbus communications using profibus networks. *Industrial Electronics, IEEE Transactions on*, 46(6):1241–1251, Dec 1999.

[11] G.C. Walsh and Hong Ye. Scheduling of networked control systems. *Control Systems Magazine, IEEE*, 21(1):57–65, Feb 2001.

[12] G.C. Walsh, Hong Ye, and L.G. Bushnell. Stability analysis of networked control systems. *Control Systems Technology, IEEE Transactions on*, 10(3):438–446, May 2002.

[13] J. Wang and Binoy Ravindran. Time-utility function-driven switched ethernet: packet scheduling algorithm, implementation, and feasibility analysis. *Parallel and Distributed Systems, IEEE Transactions on*, 15(2):119–133, Feb 2004.

[14] T.C. Yang. Networked control system: a brief survey. *Control Theory and Applications, IEEE Proceedings* -, 153(4):403–412, July 2006.

[15] Hong Ye, G.C. Walsh, and L.G. Bushnell. Real-time mixed-traffic wireless networks. *Industrial Electronics, IEEE Transactions on*, 48(5):883–890, Oct 2001.

[16] Wei Zhang, M.S. Branicky, and S.M. Phillips. Stability of networked control systems. *Control Systems Magazine, IEEE*, 21(1):84–99, Feb 2001.