An Implementation of a Networked PI Controller over IP Network

Yodyium Tipsuwan
Member, IEEE
yyt@ku.ac.th
Department of Computer Engineering,
Kasetsart University, Bangkok, Thailand, 10900

Mo-Yuen Chow
Senior Member, IEEE
chow@eos.ncsu.edu
Department of Electrical and Computer Engineering,
North Carolina State University, Raleigh, NC, USA, 27606

Rangsarit Vanijjirattikhan
Student Member, IEEE
rvanijj@unity.ncsu.edu

Abstract—This paper presents an implementation of a networked PI controller using a gain scheduling methodology with respect to real-time IP traffic conditions. The implementation enables the existing PI controller to be used over IP networks with a general network protocol like Ethernet. This paper first describes the controller design scheme using a generalized exponential distribution traffic model. The gain scheduling technique is also explained. The detail of networked PI controller implementation based on RTLinux on an actual DC motor setup is then described. Experimental results show that the proposed implementation is feasible to improve the performance of the PI controller operated over IP networks.

Index Terms—Internet, networks, adaptive control, DC motors, distributed control, real time system.

I. INTRODUCTION

A recent and advancing trend in the networked control area is to substitute specialized industrial networks with a general computer network such as Ethernet and wireless Ethernet to control a high performance application in the area of distributed control, industrial electronics, and factory automation remotely over the Internet or IP (Internet Protocol) network [1]. A general protocol like Ethernet has several advantages due to its affordability, widespread usage, and well-developed infrastructure for Internet connection. Nevertheless, once a networked control system is connected through the Internet, the network delays induced by IP are no longer constant and can vary depending on traffic conditions. Several methodologies have been developed to handle the time-varying network delay effects based on different techniques such as state augmentation [2], optimal stochastic control [3], and sampling time scheduling [4]. Many of these techniques require a completely redesigned controller to handle network delays. Practical controllers that exist in many industrial applications such as a PI (Proportional-Integral) controller have to be replaced with a new controller. This replacement is usually time-consuming, which implies high installation cost.

This paper describes an implementation of a gain scheduling scheme [5] to enable the widely used PI controller so it can be used over IP networks with a general computer protocol like Ethernet without requiring redesign and reinstallation of a new networked control system. The implementation is based on PI controller gain scheduling. The optimal PI controller gains are scheduled in real-time with respect to the monitored IP network traffic condition in order to maintain the best possible system performance. Therefore, changes in IP network traffic conditions are always captured.

II. SYSTEM DESCRIPTION

A. Mathematical Formulation

[5] has described the mathematical formulation of the networked PI controller gain scheduling. For completeness, we summarize the problem formulation in this section. In order to analyze how to schedule the PI controller gains with respect to an IP network traffic condition, let us formulate the problem mathematically in a continuous-time approach by first assuming IP network delays are constant. A typical single networked control system is formulated as shown in Fig. 1, where \( r(t), u(t), y(t), \) and \( e(t) = r(t) - y(t) \) are the reference, control, output, and error signals, respectively, which is implemented as shown in Fig. 2.

\[
\begin{align*}
  r(t) & \xrightarrow{G_p} u(t) \xrightarrow{G_c} e(t) \xrightarrow{e^{-\alpha t}} \xrightarrow{e^{-\alpha t}} y(t)
\end{align*}
\]

Fig. 1. A point-to-point networked control system formulation.

The plant dynamics is expressed as a transfer function \( G_p \), where the PI controller \( G_c \) is described by:

\[
u(t) = K_p e(t) + K_i \int_0^t e(\xi) d\xi,
\]

where \( K_p \) and \( K_i \) are the proportional gain and integral gain, respectively. The network delays for sending the control \( u(t) \) to the plant \( G_p \), and for sending the system output \( y(t) \) to the PI controller \( G_c \), are represented by \( G_{DCP}(s) \) and \( G_{IPC}(s) \), respectively, of which the analytical forms in frequency domain are:
\[ G_{DCP}(s) = e^{-\tau_{DCP} s}, \]
\[ G_{DPC}(s) = e^{-\tau_{DPC} s}, \]
where \( \tau_{DCP} \) and \( \tau_{DPC} \) are the delay from the controller to the plant, and the delay from the plant to the controller in time domain, respectively.

### B. Case Study

A DC motor speed control problem is used as a case study to demonstrate the implementation of the proposed gain scheduling described in [5] throughout this paper. The DC motor transfer function from [6] is used to represent the plant dynamics, and is described by:
\[ G_p(s) = \frac{2029.826}{(s + 26.29)(s + 2.296)}. \]

We assume that the practical DC motor PI speed controller is designed and tuned without concerning for the network delays to have the relative damping ratio of 0.707 and to satisfy the following specifications with the step response.

1. **Percentage overshoot** (P.O.): P.O. \( \leq 5\% \).
2. **Settling time** \( t_s \): \( t_s \leq 0.309 \) sec.
3. **Rise time** \( t_r \): \( t_r \leq 0.117 \) sec.

Using the root locus design approach without considering network delays, a feasible choice of \( K_p^0, K_i^0 \) can be easily evaluated.

#### III. Parameterization for Gain Scheduling: Constant Network Delay

### A. PI Controller Parameterization

With network delays in the control loop, the initial \( K_p^0, K_i^0 \) may no longer satisfy the specifications. The system performance will degrade, and the system may become unstable. To remain the best possible system performance with network delays, the controller gains need to be adapted with respect to the current network condition. In this section, we introduce the \( \beta \) gain to externally adapt \( K_p^0, K_i^0 \) without completely redesigning the existing controller. The idea of \( \beta \) gain adaptation is adopted from [7]. The \( \beta \) gain has to be greater than zero to avoid positive feedback. The \( \beta \) gain is placed in front of the initial PI controller as depicted in Fig. 3.

![Fig. 3. Adaptation of PI controller gains at the controller output by \( \beta \).](image)

Analytically, \( \beta \) adjusts both \( K_p^0 \) and \( K_i^0 \), while keeping the ratio between both gains as follows:
\[ \beta u(t) = \beta \left( K_p^0 e(t) + K_i^0 \int_0^t e(\xi) d\xi \right). \]

This parameterization enables PI gain scheduling to be tractable for real-time on-line analysis with existing theories such as root locus so that the PI controller could quickly analyze the system to perform additional advanced control schemes such as fault detection and diagnosis.

In order to evaluate the best possible system performance with respect to \( \beta \) under different IP network conditions, we minimize the following cost function to find the optimal \( \beta \):

\[ J = w_1 J_1 + w_2 J_2 + w_3 J_3, \]
\[ J_1 = \left\{ \begin{array}{ll} \frac{(MSE - MSE_0)^2}{MSE_0}, & MSE > MSE_0, \\
0, & MSE \leq MSE_0, \end{array} \right. \]
\[ J_2 = \left\{ \begin{array}{ll} (P.O. - P.O_0)^2, & P.O. > P.O_0, \\
0, & P.O. \leq P.O_0, \end{array} \right. \]
\[ J_3 = \left\{ \begin{array}{ll} (t_r - t_{r_0})^2, & t_r > t_{r_0}, \\
0, & t_r \leq t_{r_0}, \end{array} \right. \]
where
\[ MSE = \frac{1}{N} \sum_{k=0}^{N-1} e^2(k) \]
is the mean-squared error, \( MSE_0 \) is the nominal mean-squared error, \( P.O_0 \) is the nominal percentage overshoot, and \( t_{r_0} \) is the nominal rise time. The weights \( w_1, w_2, \) and \( w_3 \) are used to specify the relative significance of \( J_1, J_2, \) and \( J_3, \) respectively, on the overall system performance. The error \( e(k) = y(k) - r(k) \) is computed by sampling \( y(t) \) at \( t = kT \), where \( T \) is the sampling period, and \( k \) is the time index. The costs \( J_1, J_2, \) and \( J_3 \) are mainly used to provide the penalty when the system performance degrades from the nominal system performance. In this case, the nominal performance can be adopted from the design specifications mentioned earlier such that \( P.O_0 = 5\%, \ t_{r_0} = 0.117 \), whereas \( MSE_0 \) has to be determined from a simulation or an experiment. In this paper, we use \( MSE_0 = 0.00595 \). Therefore, when \( \beta = 1 \) without network delays in the system, \( J = 0 \).

With network delays, \( \beta = 1 \) may no longer be optimal. Thus, the optimal gain has to be obtained by evaluating \( J \) with concern for current network delays. Unfortunately, \( J \) usually does not have a closed-form relationship with \( \beta \). We define \( \mathcal{F} \) as the feasible set containing all \( \beta \) that do not cause system instability. The detail for estimating \( \mathcal{F} \) from \( \tau = \tau_{CP} + \tau_{PC} \) and finding the optimal \( \beta \) is described in [5].

### IV. Parameterization for Gain Scheduling: Actual IP Network Delay

#### A. Parameterization of IP Network Delay Characteristics for Gain Scheduling

For actual IP networks, the RTT (Round-Trip Time) delay can be treated as the delay \( \tau \). RTT delay is stochastic and happens as discrete events. To adjust the \( \beta \) gain, RTT delay is modeled by a generalized exponential distribution to describe IP network delays [5] as follows:
\[ P[\tau] = \left\{ \begin{array}{ll} \frac{1}{\phi} e^{-(\tau - \eta)/\phi}, & \tau \geq \eta, \\
0, & \tau < \eta, \end{array} \right. \]
where the expected value of the RTT delay \( E[\tau] = \phi + \eta \), and variance \( \sigma^2 = \phi^2 \). If \( \eta \) is known, \( \phi \) can be easily
approximated from $\eta$, and an experimental value of $E[\tau]$ or the mean $\mu$. In many cases, (11) can provide reasonable accuracy to represent RTT delay in real-time uses with its simple form as illustrated Fig. 4 [5]. In this paper, we treat the IP network stochastic behavior as a parameter variation of the system transfer function. Therefore, $\eta$ should be an appropriate value to serve as a base for the parameter variation described as $\tau = \eta + \Delta \tau$, where $\Delta \tau$ is the delay parameter variation. Also, based on (11), $P[\tau = \eta]$ should be the peak of the probability density function. A feasible choice of $\eta$ is the median of RTT delay. The median can be easily computed in real-time and is representative for a majority of RTT delay [5]. We ignore the case $\Delta \tau < 0$ (i.e., $\tau < \eta$) since this variation is relatively small. In addition, $P[\tau = \eta]$ could be used as the worst-case RTT delay distribution for $\tau < \eta$.

Based on (11) and the RTT delay statistics in [5], we also assume that RTT delays have $\mu > \eta$.

![Fig. 4. Comparison of (a) RTT delay histogram between ADAC (Advanced Diagnosis And Control) lab, North Carolina State University, NC to www.visitnc.com, and (b) the generalized exponential distribution shape.](image)

With the stochastic behavior of actual IP networks, $\phi$ could affect the optimal setting of the PI controller as the delay transfer functions in the control loop are changed. Thus, the optimal $\beta$ under different actual IP network traffic conditions has to be evaluated with respect to the updated $\eta$ and $\phi$. The detail about finding the optimal $\beta$ is discussed in [5].

### B. Gain Scheduling Algorithm

A possible solution to handle the variable traffic condition on actual IP networks is to adjust $(K^p, K^i)$ by scheduling $\beta$ with respect to the current traffic condition characterized by $\eta$ and $\phi$. We assume that the gain scheduling approach is implemented as a hardware or software called the $\beta$ scheduler middleware, which is physically (in hardware) or virtually (in software) attached in front of the original PI controller, respectively. The gain scheduling procedure is described as follows.

1) The $\beta$ scheduler middleware initializes the packet index defined as $i$ to 0, the summation of RTT delay defined as $m$ to 0, and the number of successful packet roundtrips defined as $n$ to 0 to be used in later steps.

2) To send $u(t)$ out to a plant according to the original controller operation, the middleware captures and puts $u(t)$ in a UDP packet at every sampling time $T$ with $i$, and the current time defined as the sending time $t_s(i)$. The control $u(t)$ in the packet $i$ is defined as $u(t, i)$ for future reference. The packet is sent out immediately if possible. However, the network may not be always available for a transmission. Thus, the packet $i$ may have to be stored in the output queue to wait for sending at the instant that the IP network is ready. Once the packet $i$ is pushed in the queue, or sent out immediately without being stored, the middleware will increase the packet index by $i = i + 1$.

3) The plant will return the output $y(t)$, $i$, and $t_s(i)$, as a packet back to the middleware once it receives and processes $u(t,i)$ periodically using the sampling period $T$. This corresponding feedback is defined as $y(t, i)$. Likewise, we assume the plant has the same queueing mechanism to handle outgoing packets.

4) When the $\beta$ scheduler middleware receives a packet containing $y(t, i)$ from the plant at time $t$ during a sampling period, the $\beta$ scheduler middleware will compute:

$$rtt(i) = t_s(i) - t_s(i),$$

$$m = m + rtt(i),$$

$$n = n + 1,$$

where $rtt(i)$ is the RTT delay of the packet roundtrip $i$, and $t_s(i)$ is the arrival time of the corresponding feedback packet $i$. The summation of $m$ is used to later compute the mean $\mu$. The middleware will store $rtt(i)$ in memory along with other RTT denoted as $rtt(j)$, $\forall j \in \mathbb{N} < i$ that are previously computed. The RTT delay $rtt(i)$ is placed in the memory, at which $rtt(a) < rtt(i) \leq rtt(b)$, $\forall a, \forall b \in \mathbb{N} < i$ for sorting RTT delays in the memory to later compute $\eta$. For future reference, the RTT delay stored at the position $l$ in the memory is defined as $RTT[l]$.

Packets transmitted between the middleware and the plant may be lost because of several reasons such as IP network congestion and a router’s packet dropping policy. Therefore, there would be some unsuccessful packet roundtrips. In this case, the PI controller and the plant will opt to use the most updated data to compute $u(t, i)$ and $y(t, i)$, respectively. In this paper, we focus on the effect of IP network delay and variation, and assume that the number of unsuccessful packet roundtrips is small such that it does not significantly affect the control performance.

5) Once $n = N$, the middleware will calculate:

$$\mu = \frac{m}{N},$$

$$\eta = \begin{cases} \left\lceil RTT\left[\frac{N}{2}\right] + RTT\left[(N/2) + 1\right] \right\rceil / 2, & N \text{ is even,} \\ RTT\left[\frac{N}{2}\right], & N \text{ is odd,} \end{cases}$$

$$\phi = \mu - \eta,$$

where $N$ is the number of packets used to approximate the characteristic of RTT delay. When $\mu \leq \eta$, $\phi = 0$ and $\eta$ becomes the representative worst-case delay to avoid a negative $\phi$, which violates the shape of (11). The middleware then updates $\beta$ by picking the optimal $\beta$ with respect to $\eta$ and $\phi$. Steps 1-5 will be repeated for the next iteration.

## V. Implementation

An actual implementation of the PI controller with $\beta$ scheduler middleware over IP networks depicted as the block diagram in Fig. 2 is shown in Fig. 5.
A. PI Controller with $\beta$ Scheduler Middleware

The PI controller with $\beta$ scheduler middleware is implemented on a PC running RTLinux 3.0. This software implementation has two parts. The first part is the hard real-time program to compute $u$ and $\beta$. The second sub part is the soft real-time program, which handles IP network communication with the remote plant. The hard real-time program runs periodically with the sampling period $T$. Both programs exchange data between each other via RTFIFO (Real-Time-First-In-First-Out) protocol, which is a special queue mechanism on RTLinux to exchange data between a hard real-time process and a normal process. The flowcharts to describe both software parts are shown in Fig. 6.

The hard real-time program contains two subparts. The first subpart computes the driving voltage for the DC motor based on the original PI controller design. The second subpart handles communication with the soft real-time program and modifies $u$ from the first subpart by the optimal $\beta$ gain.

The combination of the second subpart and the soft real-time program can be viewed as the $\beta$ scheduler middleware. The first subpart reads the motor speed $y$ if $y$ is available in RTFIFO 1 and computes the control $u$. Then, the second subpart modifies $u$ with the optimal $\beta$ value. The resultant control $u$ will be put into RTFIFO 2, and ready to be read and sent to the remote plant by the soft real-time program. The second subpart also computes and stores RTT delay for computing $\eta$ and $\phi$ values. The soft real-time program continuously monitors IP network socket connection at an arbitrary port 6789. If there is an incoming feedback UDP packet from the remote plant system, the program will extract the speed feedback $y$ from the packet and send $y$ to the hard real-time program through RTFIFO 1. Also, the soft real-time program always checks if $u$ is updated by the hard real-time program. Once $u$ is available in RTFIFO 2, the soft real-time program will transmit the control packet to the remote plant.

B. Remote Plant System

The remote plant system is composed of multiple hardware units as follows.

1) DC motor system

The DC motor model used in this paper is MT150F manufactured by Feedback. The motor is driven by a servo amplifier SA150D produced by the same company. The input voltage range of the driver for motor speed control is 0-15 V. The motor speed is measured from the tachometer on the MT150F motor. The output voltage range of the tachometer is

![Fig. 6. Hard real-time and soft real-time programs in PI controller with $\beta$ scheduler middleware](image-url)
±13.5 V.

2) Level shifter

This circuit is implemented by two NOT gates, two NAND gates, and an ICL7667 MOSFET driver as shown in Fig. 7. The level shifter amplifies the PWM signal of the C515C microcontroller from 0-5 V to 0-15 V and switches the motor direction as needed at the input of the servo amplifier.

3) Signal conditioner

The signal conditioner circuit shown in Fig. 8 is used to scale the tachometer output from ±13.5 V to 0-5 V so that the C515C microcontroller can perform A/D conversion. The circuit is comprised of four Op-Amp stages using ±15 V as the reference voltages. The first stage Op-Amp provides high impedance input to the tachometer and low impedance output to the inverting amplifier at the second stage, while remains the same voltage as from tachometer. Then, the inverting amplifier scales the signal in the range of ±13.5 V to ±2.5 V. The output of the second stage Op-Amp is buffered by the third stage Op-Amp. Finally, the signal is adjusted into the range of 0-5V by the summing Op-Amp in the fourth stage. The Burr-Brown OPA132 Op-Amp is chosen for the first and second stages because of its ability to handle high voltage. Instead, the Burr-Brown OPA27 is used in the third and fourth stages because of its low noise level and inherently low offset voltage.

4) KitCON-515C microcontroller board

The KitCON-515C microcontroller board is the Starter Kit board of the Siemens (or Infineon) C515C microcontroller. The board is utilized for two purposes.

a) Convert the motor speed \( y \) extracted from a control packet to an actual PWM signal to drive the DC motor. The PWM signal is generated by an internal timer, and is sent out to the level shifter.

b) Perform A/D conversion of the signal conditioner output. The result, which indicates the motor speed, is then sent back to the remote station.

5) Remote station

The remote station is a PC running RTLinux 3.0 to link IP networks and the motor system together via a parallel port. The station works similarly to the PI controller with \( \beta \) scheduler middleware. There are also a hard real-time program and a soft real-time program, which communicate via RTFIFO. Likewise, the soft real-time program continuously monitors the IP network socket connection at port 6789. If there is an incoming control packet, the program will extract the control \( u \) from the packet and send the data to the hard real-time program through RTFIFO 3. In addition, the soft real-time program always checks if \( y \) is updated by the hard real-time program. Once \( y \) is available in RTFIFO 4, the soft real-time program will transmit a feedback packet back to the PI controller.

The hard real-time program runs periodically with sampling period \( T \) to convert the control \( u \) from RTFIFO 3 to a PWM signal, and put the speed data from the microcontroller board into RTFIFO 4 for speed feedback transmission.

VI. EXPERIMENTAL RESULT

The performance of the proposed \( \beta \) scheduler middleware approach on the networked DC motor PI speed control system is verified by an experimental setup in addition to the simulation results presented in [5]. The following environment is used to illustrate the effectiveness of the proposed \( \beta \) scheduler middleware control scheme.

- The steady-state reference value: \( c = 50 \) rad/s.
- The final time of experimental testing: \( t_f = 5 \) sec.
- The sampling time of the PI controller, the \( \beta \) scheduler middleware, and the remote station: \( T = 40 \) msec.
- The number of packets used to evaluate the characteristic of RTT delay: \( N = 100 \).

To investigate the effectiveness of the \( \beta \) scheduler
middleware on an actual IP network, three scenarios are examined:

1. The DC motor is controlled directly by the PI controller without network delays using the nominal gains \(K_p^0, K_i^0\).

2. The DC motor is controlled over an IP network by the PI controller with the nominal gains \(K_p^0, K_i^0\). The PI controller is connected to the Internet via a DSL service in Raleigh, NC, whereas the DC motor system is in ADAC lab at North Carolina State University. Typical RTT delay measured by the middleware using \(T=40\) msec is shown in Fig. 9.

3. The DC motor is controlled over an IP network by the PI controller with \(\beta\) scheduler middleware using \(\eta\) and \(\phi\) estimated from the real-time measurements on the IP network RTT delay. The IP network connection in this scenario is the same as the second scenario.

The costs \(J\) from the experiments on the three scenarios are shown in Table 1, whereas the step responses from the experiment are illustrated in Fig. 10.

**Table 1. Costs \(J\) from Network DC Motor PI Speed Control Experiments.**

<table>
<thead>
<tr>
<th>Without delay</th>
<th>With IP network delays using the nominal gains</th>
<th>With IP network delays using (\beta) gain scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>36.8113</td>
<td>0.2402</td>
</tr>
</tbody>
</table>

An implementation of the proposed gain scheduling approach with respect to the current network traffic condition is introduced in this paper to enhance widely used and existing PI controllers for using over IP networks. The experimental results have shown the promising feasibility to apply the proposed gain scheduling scheme for a networked PI controller on actual IP network environment with reasonably long RTT delays and relatively low variations. There are still several issues to be investigated in order to improve and strengthen the gain scheduling approach such as packet loss effects, and the performance of the gain scheduling on actual networked control systems over real IP networks under various IP QoS protocols. These additional studies could support the gain scheduling approach for practical uses in the future.

**VII. Conclusion**

An implementation of the proposed gain scheduling approach with respect to the current network traffic condition is introduced in this paper to enhance widely used and existing PI controllers for using over IP networks. The experimental results have shown the promising feasibility to apply the proposed gain scheduling scheme for a networked PI controller on actual IP network environment with reasonably long RTT delays and relatively low variations. There are still several issues to be investigated in order to improve and strengthen the gain scheduling approach such as packet loss effects, and the performance of the gain scheduling on actual networked control systems over real IP networks under various IP QoS protocols. These additional studies could support the gain scheduling approach for practical uses in the future.

**Acknowledgment**

The authors would like to thank the Royal Thai Scholarship for partially supporting this study.

**References**


