Lazy Scheduling of Processing and Transmission Tasks in Collaborative Systems

Sasa Junuzovic
Computer Science Department
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA
sasa@cs.unc.edu

Prasun Dewan
Computer Science Department
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA
dewan@cs.unc.edu

ABSTRACT
A collaborative system must perform both processing and transmission tasks. We present a policy for scheduling these tasks on a single core that is inspired by studies of human perception and the real-time systems field. It lazily delays the execution of the processing task if the delay cannot be noticed by humans. We use simulations and formal analysis to compare this policy with previous scheduling policies. We show that the policy trades-off an unnoticeable degradation in performance of some users for a much larger noticeable improvement in performance of others.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems – distributed applications, client/server. C.4 [**Performance of Systems**] Performance Attributes.

General Terms
Algorithms, Measurement, Performance, Experimentation.

Keywords
Collaboration architecture; scheduling policy; local and remote response times; analytical model; simulations.

1. INTRODUCTION
In a collaborative system, a computer must not only process user commands but also transmit these commands to the computers belonging to other users. How the processing and transmission tasks are scheduled can influence the response times seen by local and remote users, which we refer to as local and remote response times, respectively. One approach is to create separate threads for these tasks and schedule the threads using a round-robin policy. An alternative to this concurrent policy is to schedule the tasks in a serial order, either processing or transmitting first. Processing first tends to give the best local response times, transmitting first tends to give the best remote response times, and concurrent execution tends to give response times that are in between those supported by the other two policies [5]. Moreover, these differences can be significant in certain scenarios when a single core is available for executing these tasks [5].

One issue with the three existing policies is that there is no way to control the tradeoff between local and remote response times. Controlling the tradeoff is particularly attractive when an unnoticeable increase in one metric can result in a noticeable decrease in the other metric. Human-perception studies have shown that certain increases are indeed unnoticeable – users cannot distinguish between local response times below 50ms [9] and visual and haptic remote response times below 50ms and 25ms, respectively [4]. On the other hand, users can notice 50ms changes in local [11] and remote response times [4]. Based on these observations and the principles in real-time scheduling, we have devised a new lazy process-first scheduling policy that, like the process-first policy, gives precedence to the processing task, but delays its execution if the resulting increase in local response times cannot be noticed by humans. We can imagine a dual of this policy, lazy transmit-first scheduling, that like the transmit-first policy, gives precedence to the transmission task, but delays its execution if the resulting increases in remote response times are not noticeable. In this first cut at the idea of lazy scheduling of collaboration tasks, we have only considered lazy process-first scheduling, which we shall refer to as simply lazy scheduling.

The general idea of scheduling tasks so that they complete "just in time" is not new. In real-time systems, for example, there are both real-time tasks, which need to complete within some absolute deadline, and non-real-time tasks, which need to complete as soon as possible. Since in these systems a) there is no benefit to completing a real-time task before its deadline and b) real-time tasks typically have processing times which are shorter than their deadlines, a real-time task can be delayed by the difference between its deadline and processing time while still being able to complete in time. The amount the task can be delayed is called slack time. To improve the performance of non-real-time tasks, some slack-stealing scheduling algorithms schedule the non-real-time tasks during the slack times of the real-time tasks. By doing so, the non-real-time tasks can complete earlier.

The idea of lazy scheduling, however, raised three questions that have not been answered before. (1) How is it implemented in various kinds of distributed collaborative systems that exist today? Previous algorithms devised for supporting slack stealing cannot be used because neither the processing nor the transmission tasks have so far been modeled as tasks with deadlines. (2) How do the response times of lazy scheduling compare to those supported by the three existing policies in arbitrary theoretical scenarios? (3) Are there realistic scenarios in which it offers significant advantages over the other policies? The rest of this paper addresses these questions.

2. IMPLEMENTATION

Before we can address scheduling of the processing and transmission tasks, we must understand the nature of these tasks, which depends on the processing and communication architecture.

2.1 Processing architecture

Two popular processing architectures are the centralized and replicated architectures. In both cases, it is assumed that the shared application is logically divided into separate user-interface and processing components. The user-interface component transforms user input into input commands and sends these commands to the program component. Conversely, it processes output commands that it receives from the program component and transforms the result into updates to the display. The program component processes user input by converting input commands to output commands. The user-interface component is replicated on each user's computer and allows a user to manipulate application state not shared with the other users. The program component is logically shared by all users and may be physically centralized or replicated, depending on the processing architecture. Each user interface is mapped to exactly one program component.

In the centralized (client-server) architecture, all of the user-interface components are mapped to a single program component running on one of the user's computers. The computer running the program component is called the master and all of the other computers are called slaves. In the replicated (peer-to-peer) architecture, each user-interface component is mapped to the program component running on the local computer. Whenever a master receives an input command from the local user, it sends the command to all of the other computers, thereby ensuring the program components on different masters are kept in sync.

2.2 Communication architecture

A master computer may use unicast or multicast to communicate with other computers. The idea of multicast requires the construction, for each source of messages, a multicast overlay that defines the paths a message takes to reach the destinations. In this paper, we make two assumptions regarding multicast. First, because IP-multicast is not widely deployed, we assume an application-layer multicast in which end-hosts form the overlay. Second, as in peer-to-peer file sharing systems, we assume that only the users' computers can be used in the overlay.

When considering communication at the application layer, it is important to use application-layer transmission costs. Traditionally, the transmission time of a command has been calculated as "size of message/bandwidth." However, this calculation is invalid at the application layer because it accounts only for the transmission costs at the network interface [2]. Before the command reaches the network interface, the operating system must traverse the network stack and copy data buffers along the way, which takes time. Moreover, the operating system must perform these steps for each destination. Study by Abdelkhalek et al. [1] of the server for Quake, a popular multi-player first-person shooter game, found that these costs can be significant in practice. They found that the server spent 50% of CPU time on transmitting commands to clients. Hence, from here on, by transmission costs, we mean application-layer transmission costs. To ensure good local response times of slave users, we assume that in a centralized architecture, the master transmits an output command to the inputting slave first regardless of whether unicast or multicast is used.

2.3 Lazy Scheduling Algorithm

Our lazy scheduling algorithm works for both the centralized and replicated processing architectures and unicast and multicast communication architectures. It delays the execution of the processing task on a computer without allowing the local user to notice the delay. The pseudo-code for the algorithm is shown in Figure 1. As the figure shows, the algorithm accepts two parameters, namely, the local and remote response time degradation thresholds (Line 0). The algorithm supports different values of these thresholds because, as mentioned above, previous work has shown that noticeable local and remote response time degradations can be different.

The starting point of the policy, the `Main` function, is a loop which waits for the next command, `C`, which may be received from the local user or a remote computer. The first task is to compute from `C` the command `CtoTrans` to be actually transmitted to other computers (Lines 3-5). In all scheduling policies, this processing subtask is never delayed as it is necessary to define the transmission task. If the centralized architecture is being used and `C` is an input command, then the transmitted command is the output command corresponding to the received command; otherwise it is simply the received command.

The next step is to compute the amount of time, `maxTransTime`, by which the (remaining) processing can be delayed, which depends on whether `C` is entered by or is a response to a command entered by the local or remote user (Lines 8-10). When `C` is an input command from the local user, then the processing of the command can be delayed by as much as the time specified in the local response time degradation threshold. Hence, `maxTransTime` is set to this threshold. When `C` is an output to a command entered by the local user, however, the command can be delayed for some time at the master for reasons given later. The delay is stored in `prevDelay` property of the command. Thus, `maxTransTime` is set to the difference between the local response time threshold and `prevDelay`. In all other cases, processing can be delayed by as much as the remote response time degradation threshold. Since the computers on the path from the source to the current computer contribute to the remote response times of the computer, the processing can be delayed only if the total previous delay for the command, which is stored in `prevDelay`, is less than the remote response time threshold. Thus, `maxTransTime` is set to the difference between this threshold and the total previous delay of the command.

Once `maxTransTime` has been calculated, the algorithm calls the `Transmit` function (Line 11). The `Transmit` function returns if it estimates that it will execute for longer than `maxTransTime` if it transmits to another destination (Line 16). To approximate the cost of the next transmission, the `Transmit` calls the `EstTransTime` function. To provide the estimate, the `EstTransTime` function could use data from previous collaborations or can dynamically determine transmission costs

```
0: INPUT: Local (L) and remote (R) response time
          degradation thresholds
DESTS      // this computer's destinations
Main()
1: loop (forever)
2:  wait for command C
3:  if(centralized && C.isInput)
4:    CtoTrans = Process(C)
5:  else CtoTrans = C
6:  startTime = now
7:  for(each dest in DESTS) dest.sentTo = false
8:  if((C.isInput && C.isFromLocalUser) ||
      (C.isOutput && C.isOutputToCmdByLocalUser))
9:    maxTransTime = L - C.prevDelay
10: else maxTransTime = R - C.prevDelay
11: Transmit(CtoTrans,maxTransTime)
12: if(centralized) Process(CtoTrans)
13: else Process(Process(C))
14: Transmit(CtoTrans,INIFINITY)
Transmit(CtoTrans,maxTransTime)
15: for(each dest in DESTS)
16:  if(now - startTime + EstTransTime(CtoTrans)
          >= maxTransTime) return
17:  if(dest.sentTo == false)
18:    CtoTrans.prevDelay +=
         now - startTime + EstTransTime(CtoTrans)
19:    dest.send(CtoTrans)
20:    dest.sentTo = true
```

Figure 1. The lazy scheduling policy algorithm.

based on transmission times of previous commands. If the `Transmit` function does not return, it transmits the command to the next destination. Because it can be called twice for the same command, once before and once after the processing task is performed, the function keeps track of destinations to which it has already sent the command and does not send to those destinations again when it is called the second time (Lines 17-20). For each transmitted command, it stores the delays the command has experienced so far in `prevDelay` (Line 18), which the next computer on the path reads (Lines 8-10).

After the `Transmit` method is called the first time, the processing of the command completes at the local computer (Lines 12-13), which depends on the processing architecture. In the centralized case, only the output command is processed. In the replicated case, the input and its computed output are processed. Then, the `Transmit` method is called again, this time with `maxTransTime` set to `INIFINITY`, which allows the transmission to complete.

We have presented the algorithm with respect to a single input command for single-core machines. We defer the issues of multi-core scheduling, simultaneous commands, and concurrency control to the discussion section.

3. ANALYSIS

We evaluate our lazy algorithm by comparing it with the existing sequential and concurrent policies. We present and illustrate equations for the response times for the four policies, and use the equations to predict the relative performances of the policies.

3.1 Replicated Remote Response Time

We first develop the equations for replicated architecture remote response times for input commands entered by a master user. To reach a particular user's computer, which we refer to as the destination computer, the command must travel from the source computer to the destination computer along some path. The path may consist of additional computers, which we refer to as intermediate computers. The terms destination and intermediate are relative to a particular path. An intermediate computer on one path is a destination computer on a different path as all users see the output of an input command. Let π denote the path from the source to the destination, m denote the number of computers on the path including the source and destination computers, and $\pi_k, 1 \le k \le m$, denote the k^{th} computer on the path π, where π_1 is the source and π_m the destination computer.

The replicated remote response time of command i to computer j along path π is given by

$$remote_{REP,i,j} = \sum_{k=1}^{m-1} d\left(\pi_k, \pi_{k+1}\right) + \sum_{k=1}^{m-1} \delta\left(\pi_k\right) + \beta\left(\pi_m\right)$$

where $d\left(\pi_k, \pi_{k+1}\right)$ is the network latency between the k^{th} and $k+1^{st}$ computers on path π, $\delta\left(\pi_k\right)$ is the delay of the k^{th} intermediate computer on the path, and $\beta\left(\pi_m\right)$ is the delay of the destination computer. The destination and intermediate computers contribute different delays because the former contributes to the remote response time of the local user while the latter contribute to the remote response time of a remote user. This results in a fundamental difference between the equations for the intermediate and destination computers. In the case of an intermediate computer, we must determine when the computer transmits to the downstream computer. In the case of the destination computer, we must determine when the input and output processing complete.

The first component of the remote response time equation is independent of the scheduling policy as it is a sum of the network latencies on the path from the source to the destination.

3.1.1.1 Transmit-first Policy Delays

The simplest equations for the computer delays along the path π from the source to the destination are those for the transmit-first policy. Consider first the delay of π_k, the k^{th} intermediate computer on path π. Its delay is equal to the time that it requires to transmit the command to the next computer along the path, π_{k+1}. In general, computer π_k may have to transmit to more than one destination. Therefore, its delay depends on the number of other computers to which it transmits before transmitting to computer π_{k+1}. Let $x_{i,a}^{IN}$ denote the time user$_a$'s computer requires to transmit the input command i to a single destination, and $\sigma\left(a,b\right)$ denote the position of user$_b$'s computer in user$_a$'s computer list of destinations. Then, the transmit-first delay of the k^{th} computer on the path from the source to the destination equals

$$\delta_{REP}^{TF}\left(\pi_k\right) = \sigma\left(\pi_k, \pi_{k+1}\right) * x_{i,\pi_k}^{IN}$$

The delay of the destination computer π_m also depends on the number of computers to which it forwards commands because it must first transmit the command to all of them before processing the input command and its output. Let $p_{i,a}^{IN}$ ($p_{i,a}^{OUT}$) denote the time user$_a$'s computer requires to process input (output) command i, and let s_a denote the number of destinations to which user$_a$'s computer forwards commands. Thus, the transmit-first delay of the destination computer equals

$$\beta_{REP}^{TF}\left(\pi_m\right) = s_{\pi_m} * x_{i,\pi_m}^{IN} + p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT}$$

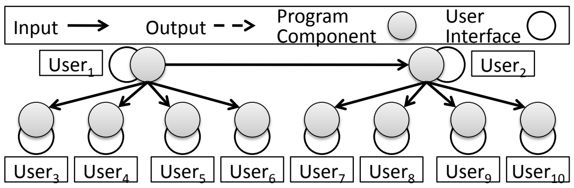

Figure 2. Replicated-multicast architecture with ten users.

To illustrate the remote response time equations for this policy, consider the replicated-multicast architecture shown in Figure 2. The figure shows the transmission of an input command entered by user$_1$. User$_1$'s computer transmits only to computers belonging to user$_2$, user$_3$, user$_4$, user$_5$, and user$_6$, while user$_2$'s computer transmits to computers belonging to user$_7$, user$_8$, user$_9$, and user$_{10}$. Suppose the architecture has the following additional properties: 1) user$_1$ enters all of the commands; 2) user$_1$'s computer's transmission order is user$_2$, user$_3$, user$_4$, user$_5$, and user$_6$, and user$_2$'s transmission order is user$_7$, user$_8$, user$_9$, and user$_{10}$; 3) all of the users have the same computers; 4) the time the computers require to process an input and output command are 3P and P, respectively; 5) the time the computers require to transmit an input command to a single destination is T; 6) that P is much greater than T; 7) the network latency between all of the computers is D; and 8) the response degradation thresholds are both T. We will use this *theoretical example* as a running example for illustrating our response time equations.

Consider the remote response time of user$_{10}$. The path π from user$_1$'s to user$_{10}$'s computer is of length $m = 3$ and π_1, π_2, and π_3 are user$_1$'s, user$_2$'s, and user$_{10}$'s computers, respectively. User$_1$'s delay is equal to the time user$_1$'s computer requires to transmit the command to a single destination, T, since it transmits to user$_2$'s computer first. Similarly, user$_2$'s delay is equal to the time user$_2$'s computer requires to transmit the command to four destinations, 4T, since it transmits to user$_{10}$'s computer last. User$_{10}$'s computer's delay is equal to the time the computer requires to process the input and the corresponding output command, 4P. But if user$_{10}$'s computer had to also forward the command to other computers, then the delay would include the time the computer requires to transmit the input command to them. Therefore, user$_{10}$'s remote response time is equal to T+4T+4P=5T+4P.

3.1.1.2 *Process-first Policy Delays*

The equations for the process-first policy delays of the computers on the path π from the source to the destination are similar. Recall that in this policy, the computer starts the transmission task after it completes the processing task. Therefore, unlike the transmit-first delay, the process-first delay on an intermediate computer includes the time the computer requires to process the input and the corresponding output command. The process-first delay of an intermediate computer π_k is given by

$$\delta_{REP}^{PF}(\pi_k) = p_{i,\pi_k}^{IN} + p_{i,\pi_k}^{OUT} + \sigma(\pi_k, \pi_{k+1}) * x_{i,\pi_k}^{IN}$$

In our example, user$_1$'s and user$_2$'s delays are equal to the time their computers require to process the input and output command, 4P, plus the time they require to transmit the input to one and four destinations, T and 4T, respectively.

The process-first delay of the destination π_m computer is simply the time the computer requires to process the input and the corresponding output command.

$$\beta_{REP}^{PF}(\pi_m) = p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT}$$

3.1.1.3 *Concurrent Policy Delays*

The delay equations for the concurrent policy are more complicated. The reason is that when the processing and transmission tasks execute concurrently, they interfere with each other's execution times. We assume that neither task blocks because it is difficult to predict their behavior, otherwise. The non-blocking task assumption is consistent with assumptions made in real-time systems when tight performance bounds are required. While results exist for blocking tasks, the upper-bounds for the performance in this case are extremely loose. Moreover, the non-blocking task assumption is realistic as a well-designed application can help ensure that the processing and transmission tasks do not block by using separate threads and asynchronous communication, respectively. In addition, we consider context switch times negligible as we have found that they are no more than a few microseconds on modern operating systems running Pentium 4 desktops, which is several orders of magnitude lower than processing and transmission costs we have observed in real collaboration scenarios. Given these assumptions and our earlier assumption that a single core is available for scheduling, then the time required to complete 1) the shorter of the transmission and processing tasks is equal to twice as long as the time required to complete it standalone and 2) the longer of the two tasks is equal to the time required to complete the two tasks sequentially. Both of these results are illustrated in Figure 3, which shows two tasks executing concurrently. As the figure shows, as the length of the scheduling quantum becomes smaller, the time required to complete the shorter task doubles. On the other hand, the longer task always completes in the amount of time required to complete the two tasks separately. This result is captured by the function

$$conc(a,b) = \begin{cases} 2a \text{ } if \text{ } a \le b \\ a + b \text{ } if \text{ } a > b \end{cases}$$

where a and b are execution times of the two tasks. Thus, we can state the concurrent delay of an intermediate computer π_k as

$$\delta_{REP}^{CONC}(\pi_k) = conc\left(\sigma(\pi_k, \pi_{k+1}) * x_{i,\pi_k}^{IN}, p_{i,\pi_k}^{IN} + p_{i,\pi_k}^{OUT}\right)$$

In our example, user$_2$'s requires 4T time to transmit the command to user$_{10}$'s computer and 4P time to process the command and its output. Therefore, user$_2$'s delay is equal to 8T since the transmission time is shorter than the processing time.

The delay of the destination computer π_m is similar except that it captures how long it takes to complete the processing task rather than how long it takes to complete the transmission task. Thus, the concurrent delay of the destination computer is

$$\beta_{REP}^{CONC}(\pi_m) = conc\left(p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT}, s_{\pi_m} * x_{i,\pi_m}^{IN}\right)$$

In our example, since user$_{10}$'s computer does not forward commands, the time it takes to complete the transmission task is 0. Therefore, its delay is equal to its processing time, 4P.

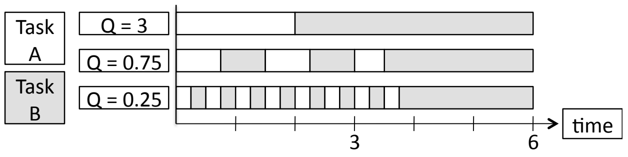

Figure 3. Concurrent execution time.

3.1.1.4 Lazy Policy Delays

The equations for the lazy policy delays of the computers along the path π from the source to the destination must account for the local and remote response time degradation thresholds.

Consider the delay of an intermediate computer π_k. This delay depends on whether the computer transmits to the next computer on the path, π_{k+1}, before or after the processing task. This, in turn depends on the difference between the sum of the delays the command has experienced so far and the amount of time by which the computer can delay the processing task without the local user noticing the delay, which is the local (remote) response time degradation threshold if the computer is (not) the source. Let

$$\tau_a = \begin{cases} \text{local response time threshold if } a \text{ is the source} \\ \text{remote response time threshold otherwise} \end{cases}$$

$$\gamma_k^{TOT} = \sum_{l=1}^{k-1} \delta(\pi_l)$$

Then, the lazy delay of an intermediate computer π_k is

$$\delta_{REP}^{LAZY}(\pi_k) = \begin{cases} \sigma(\pi_k,\pi_{k+1}) * x_{i,\pi_k}^{IN} \text{ if } \sigma(\pi_k,\pi_{k+1}) * x_{i,\pi_k}^{IN} < \tau_{\pi_k} - \gamma_k^{TOT} \\ \sigma(\pi_k,\pi_{k+1}) * x_{i,\pi_k}^{IN} + p_{i,\pi_k}^{IN} + p_{i,\pi_k}^{OUT} \text{otherwise} \end{cases}$$

In our theoretical example, recall that we assumed that both the local and remote response time degradation thresholds are T. Since the time user$_1$'s computer requires to transmit to a single destination is T, then according to the lazy policy algorithm, user$_1$'s computer will transmit to user$_2$'s computer before performing the processing task. Therefore, user$_1$'s delay is equal to T. On the other hand, user$_2$'s computer will immediately begin performing the processing task in order to meet the remote response time threshold because the command has already been delayed by T. Therefore, user$_2$'s delay is equal to the time user$_2$'s computer requires to process the input and output command, 4P, plus the time it requires to transmit the command to four destinations, 4T.

The delay on the destination computer is slightly different as it tries to forward the command to as many other computers as possible while satisfying the remote response time threshold. Hence, if the time the computer requires to complete the transmission task plus the amount of time the command has already been delayed is less than the amount of time by which the computer can delay the processing time without the local user noticing the delay, then it will complete the transmission task before performing the processing task. Otherwise, if the amount the command had been delayed so far is less than the remote response time threshold, then the computer will transmit for only the difference between the two before performing the processing task. Otherwise, it will perform the processing task immediately. Hence, the delay of the destination computer is

$$\beta_{REP}^{LAZY}(\pi_m) = \begin{cases} s_{\pi_m} * x_{i,\pi_m}^{IN} + p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT} \text{ if } s_{\pi_m} * x_{i,\pi_m}^{IN} \le \tau_{\pi_m} - \gamma_m^{TOT} \\ \max\left(0, \tau_{\pi_m} - \gamma_m^{TOT}\right) + p_{i,\pi_m}^{IN} + p_{i,\pi_m}^{OUT} \text{otherwise} \end{cases}$$

In our example, since user$_{10}$'s computer does not forward the command to other computers, its delay is the processing time, 4P.

3.2 Replicated Local Response Time

So far, we have presented only the equations for the replicated remote response times. We next present the equations for replicated local response times for commands entered by a master user. Recall that the local response time is the time that elapses from the moment a user enters an input command to the moment the user sees the output for the command, which is equivalent to the time that elapses from the moment the inputting user's computer receives the command to the moment the computer completes processing the output of the command. Therefore, the local response time is exactly the delay of the destination computer defined above. This makes sense because the inputting user's computer is both the source and the destination. Thus, the transmit-first, process-first, concurrent, and lazy local response time equations for command i entered by user$_j$ are given by

$$local_{REP,i,j}^{TF} = \beta_{REP}^{TF}(j) = s_j * x_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT}$$

$$local_{REP,i,j}^{PF} = \beta_{REP}^{PF}(j) = p_{i,j}^{IN} + p_{i,j}^{OUT}$$

$$local_{REP,i,j}^{CONC} = \beta_{REP}^{CONC}(j) = conc\left(p_{i,j}^{IN} + p_{i,j}^{OUT}, s_j * x_{i,j}^{IN}\right)$$

$$local_{REP,i,j}^{LAZY} = \beta_{REP}^{LAZY}(j) = \begin{cases} s_j * x_{i,j}^{IN} + p_{i,j}^{IN} + p_{i,j}^{OUT} \text{ if } s_j * x_{i,j}^{IN} \le \tau_j \\ \tau_j + p_{i,j}^{IN} + p_{i,j}^{OUT} \text{otherwise} \end{cases}$$

3.3 Other Cases

The equations we have presented have considered the case in which the processing architecture is replicated and the input command is entered by a master user. Let us next consider the centralized architecture and slave commands.

3.3.1 Centralized Architecture

We can obtain the centralized architecture equations for commands entered by master users from the above replicated architecture equations by adjusting them for the two main differences in the two architectures. First, in the centralized architecture, only the master computer processes input commands, while all computers process output commands. Therefore, when calculating the delays of the computers on the path from the source to the destination, the processing times in the delays are equal to the time needed to process only output commands. Second, instead of transmitting input commands, the computers transmit output commands. Based on these two differences, the centralized architecture general local and remote response time equations are, respectively, given by

$$local_{CENT,i,j} = p_{i,j}^{IN} + \beta(j)$$

$$remote_{CENT,i,j} = p_{i,\pi_1}^{IN} + \sum_{k=1}^{m-1} d(\pi_k,\pi_{k+1}) + \sum_{k=1}^{m-1} \delta(\pi_k) + \beta(\pi_m)$$

The new term accounts for the fact that the master computer must still process the input command. This is the equation that applies to all scheduling policies.

Next we can derive the policy-specific delays created by the computers on the path from a source to a destination. The centralized process-first delay on an intermediate computer π_k is

$$\delta_{CENT}^{PF}(\pi_k) = p_{i,\pi_k}^{OUT} + \sigma(\pi_k,\pi_{k+1}) * x_{i,\pi_k}^{OUT}$$

The remaining delays can be derived similarly.

3.3.2 Slave Commands

We can also obtain the equations for input commands entered by slave users by morphing the above equations for input commands entered by master users. The only difference between the two kinds of input commands is that a command entered by a slave must first reach the master computer. Once the command reaches the master, the problem reduces to that of calculating the remote response time from the master to the slave, which we have already done above. The time the command takes to reach the master computer is equal to the time the slave computer requires to transmit the command to a single destination (i.e. the master) plus the time the command takes to traverse the network between the slave and master computers. Therefore, we can obtain the equations for the local and remote response time of command i entered by slave user$_a$ whose master is user$_b$ by adding the term $x_{i,a}^{IN} + d(a,b)$ to the response time equations.

In the lazy scheme described so far, a master computer does not noticeably degrade the (local or remote) response time of its local user. It should also not noticeably degrade the local response time of a slave user. Recall that when a slave user inputs a command, the master transmits the output to that slave first. Moreover, we have found that the cost of transmitting to a single destination is typically less than the reported response time degradation thresholds of 50ms. Thus, the master will transmit the output to the slave before doing local processing of the output, which means that the delay is equal to the time the master requires to transmit the output to a single destination. Since the delay is also less than the local response time threshold, the slave's local response time does not degrade by more than the local response time degradation threshold.

3.4 Implications

The analysis above helps us better understand the nature of lazy scheduling and how it differs from the other scheduling policies. It also helps us formally confirm intuitive expectations and, more interesting, derive some unintuitive results about the lazy policy.

The lazy policy takes slack time in processing to transmit commands to other destinations. Thus, it gives processing less priority than process-first and more priority than concurrent and transmit-first policies. Intuitively, processing early (late) favors local (remote) response times, so, this seems to imply that, in comparison to (a) process-first, the local should be worse and remote response time better and (b) concurrent and transmit-first, the local should be better and remote response time worse. Our equations show that the differences are more subtle because the algorithm is run on each computer. Because of lack of space, we show the differences only for the replicated architecture and master commands. The results, however, apply to other cases also.

3.4.1 Lazy vs. Process-first

The difference in the lazy and process-first local response times is

$$local_{REP,i,j}^{LAZY} - local_{REP,i,j}^{PF} = \min\left(s_j * x_{i,j}^{IN}, \tau_j\right)$$

By definition, the difference is never more than the local response time degradation threshold, τ_j. Thus, the equations predict that the lazy local response times are never noticeably worse than the process-first local response times.

To illustrate, consider user$_1$'s local response time in our example. When the lazy policy is used, user$_1$'s computer delays the processing task by the local response time threshold, T. Thus, since the computer requires 4P time for the processing task, user$_1$'s local response time is equal to T+4P. With the process-first policy, the computer immediately processes the command, resulting in a local response time of 4P to user$_1$. Thus, the lazy local response time is worse than that of the process-policy, but not by more than the local response time degradation threshold.

While the difference in local response times is expected from the design of the lazy policy, the difference in the remote response times are somewhat surprising. Consider first an intermediate computer. If the accumulated delays are such that this computer processes the command before transmitting to the downstream computer, then the difference in the delays is given by

$$\delta_{REP}^{LAZY}(\pi_k) - \delta_{REP}^{PF}(\pi_k) = 0$$

The reason is that in this case the lazy policy behaves like the process-first policy. In the other case, when the computer transmits to the downstream computer before processing, the difference in the delays is given by

$$\delta_{REP}^{LAZY}(\pi_k) - \delta_{REP}^{PF}(\pi_k) = -\left(p_{i,\pi_k}^{IN} + p_{i,\pi_k}^{OUT}\right)$$

Let us now consider the destination computer. If the destination computer does not delay processing, the lazy policy reduces to process-first. Hence, in this case

$$\beta_{REP}^{LAZY}(\pi_m) - \beta_{REP}^{PF}(\pi_m) = 0$$

Otherwise, the difference in the delays is

$$\beta_{REP}^{LAZY}(\pi_m) - \beta_{REP}^{PF}(\pi_m) = \min\left(s_{\pi_m} * x_{i,\pi_m}^{IN}, \tau_{\pi_m} - \gamma_m^{TOT}\right)$$

The result can never be more than the remote response time threshold by definition. Since processing costs can be significant, the lazy delays on intermediate computers can be significantly better than the process-first delays. Therefore, the lazy policy remote response times can be significantly better than the process-first remote response times.

To illustrate, consider user$_{10}$'s remote response time in our example. In this case, user$_1$'s and user$_2$'s computers are the intermediate computers. User$_1$ and user$_2$ delays are equal to T and 4P+4T, respectively, with the lazy policy, and 4P+T and 4P+4T, respectively, with the process-first policy. User$_{10}$'s delays are 4P because user$_{10}$ does not forward commands. Thus, user$_{10}$'s remote response time is 4P less with the lazy than with the process-first policy. If P is large, the difference is significant.

These results show some fundamental differences between lazy and process-first scheduling. The intermediate computers on the path that delay processing have an additive effect on the improvement in the remote response time. On the other hand, the destination computer does not degrade the remote response time by more than the remote response time threshold. Thus, like local response times, lazy remote response times can never be noticeably worse than those of process-first. More important, if processing costs are high, an unnoticeable increase in the remote response time of each intermediate computer that delays processing can result in a noticeable decrease in the remote response time of the destination computer.

3.4.2 Lazy vs. Transmit-first

The differences in the local response times of the lazy and transmit-first policies is given by

$$local_{REP,i,j}^{LAZY} - local_{REP,i,j}^{TF} = \min\left(s_j * x_{i,j}^{IN}, \tau_j\right) - s_j * x_{i,j}^{IN}$$

By definition, the difference is always less than or equal to 0. When the total transmission time at the source is high, then the difference is also large. Hence, the lazy local response times can be significantly better than the transmit-first local response times, as one would expect intuitively.

To illustrate, in our example, user$_1$'s local response time is equal to T+4P with the lazy policy since T is the local response time threshold. With the transmit-first policy, the local response time is equal to 5T+4P since user$_1$'s computer transmits to five destinations. Thus, the lazy local response time is 4T less than the transmit-first local response time. Since the local response time threshold is T, the lazy local response time is significantly less.

The remote response time comparisons for an intermediate node are also as expected. If the node processes before transmitting to the downstream computer, the difference in the delay is

$$\delta_{REP}^{LAZY}(\pi_k) - \delta_{REP}^{TF}(\pi_k) = p_{i,\pi_k}^{IN} + p_{i,\pi_k}^{OUT}$$

Thus, in this case, transmit-first performs better, and the difference can be noticeable. However, if the intermediate node delays processing, the delays for the two policies are identical

$$\delta_{REP}^{LAZY}(\pi_k) - \delta_{REP}^{TF}(\pi_k) = 0$$

The results are more interesting when we consider the destination. If it does not delay processing, then the delay difference in is

$$\beta_{REP}^{LAZY}(\pi_m) - \beta_{REP}^{TF}(\pi_m) = -\left(s_{\pi_m} * x_{i,\pi_m}^{IN}\right)$$

Thus, if the sum of the processing times on all of the intermediate computers on the path from the source to the destination is greater than the total transmission time of the destination computer, then the transmit-first remote response times are better than those of the lazy policy. If the destination delays processing, on the other hand, then the difference in the delay is given by

$$\beta_{REP}^{LAZY}(\pi_m) - \beta_{REP}^{TF}(\pi_m) = \min\left(s_{\pi_m} * x_{i,\pi_m}^{IN}, \tau_{\pi_m} - \gamma_{i,m}^{IN}\right) - s_{\pi_m} * x_{i,\pi_m}^{IN}$$

By definition, the difference is always less than or equal to 0. When it is less than 0 and the intermediate delay difference is equal to 0, then the lazy remote response time will be better than the transmit-first remote response time. Hence, remote response times to some users can be better with the lazy than with the transmit-first policy, even through transmit-first policy gives higher priority to the transmission task.

To illustrate, consider user$_2$'s remote response time in our example. User$_1$'s transmit-first and lazy delays are both equal to T because in both cases, user$_1$'s computer transmits to user$_2$'s computer before processing. User$_2$'s lazy delay is equal to 4P since user$_2$'s computer immediately processes. On the other hand, user$_2$'s transmit-first delay is equal to 4T+4P since user$_2$'s computer transmits to four other computers. Thus, user$_2$'s lazy remote response time is 4T less than the transmit-first remote response time. Again, since the remote response time threshold is T, the lazy remote response time is significantly less.

3.4.3 Lazy vs. Concurrent

By using the same difference-based analysis as above, we can show that the lazy 1) local response times can be significantly better and never noticeably worse and 2) remote response times are sometimes better and sometimes worse than those of the concurrent policy. Thus again, even though the lazy policy gives higher priority to the transmission task compared to the concurrent policy, the lazy remote response times can be better.

4. SIMULATIONS

While our theoretical analysis makes some unintuitive predictions, it is not clear whether these predictions have any practical significance. Addressing this issue requires us to obtain realistic values of the equation parameters. For most parameters, such as processing and transmission costs, values can be obtained by simply performing measurements on realistic collaborative applications. The exception is the path between a source and destination computer, which depends on the multicast tree, which, in turn, requires us to simulate the algorithm we use to build it. Thus, we cannot simply plug in the values of parameters in the equations and use simulations to make the performance comparisons. We next describe how we chose the parameter values and simulated the multicast overlay.

4.1 Parameter Values

To perform meaningful simulations we need realistic values for the parameters that influence the performance of the four scheduling policies: (a) input and output processing and transmission costs; (b) the number of users; (c) the types of the users' computers; and (d) the network latencies.

To obtain realistic input and output processing and transmission costs, we identified user-commands in logs of actual application use and measured the costs of these commands. We logged two different applications: PowerPoint and a collaborative Checkers game in which users play as a team against the computer. We have space to talk about the results with only one them. We have focused on PowerPoint as it is perhaps the most popular business collaborative application today. Nonetheless, Checkers is also important as it represents a computation-intensive game watched, potentially, by a very large audience.

We analyzed recordings of two PowerPoint presentations. These recordings contain actual data and users' actions – PowerPoint commands and slides. We assumed that the data and users' actions in the logs are independent of the number of collaborators, the processing powers of the collaborators' computers, and network latencies. PowerPoint turned out to be a good choice of an application for which to analyze actual logs for two reasons: 1) the parameter values we measured in the associated logs were fairly wide spread and 2) it is frequently used in collaborations.

To obtain the processing and transmission time parameter values, we created a collaborative session with several computers on the same LAN. We designated one of the computers as the source of the commands, and then we replayed the PowerPoint logs using a Java-based infrastructure that has facilities for logging and replaying commands. We measured the processing and transmission times on the source computer. We used a P3 866MHz desktop and a P4 2.4 GHz desktop as sources, both of which were running Windows XP. The P3 desktop is used to simulate next generation mobile devices. We recorded the average amortized input and output command transmission times of each machine for PowerPoint. We removed any "outlier" entries from the average calculation, caused for instance, by operating system process scheduling issues. To reduce these issues, we removed as

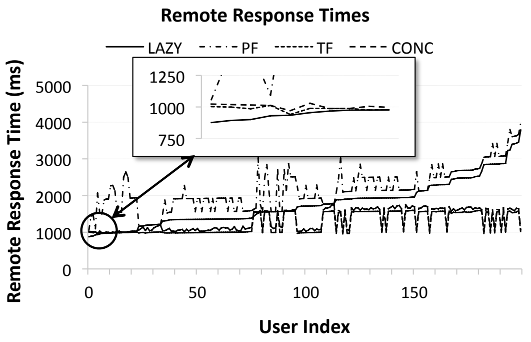

Figure 4. Remote response times for PowerPoint simulation.

many active processes on each system as possible. Ideally, while we replay the recordings, we should run a set of applications users typically execute on their systems. However, the typical working set of applications is not publicly available so we would have to guess which applications to run. For fear of incorrectly affecting our measurements by running random applications, we used a working set of size zero, a common assumption in experiments comparing alternatives.

We had to assign the values of the number of collaborators and the processing powers of their machines. In the collaboration recordings that we analyzed, the number of users ranged from thirty to sixty. Unfortunately, this is not a wide enough range of values; in particular, the maximum value of the parameter needs to be much bigger to be representative of large collaborations, such as a company-wide PowerPoint presentation. Therefore, we chose synthetic but not unrealistic values for the number of observers. As observers do not input commands, they do not influence the logs. Moreover, the talks we observed had tight time constraints which did not allow questions. Thus, they were independent of the number of observers. We randomly assigned the type of computer of each observer to be a P3 or P4 desktop.

Based on pings done on two different LANs, we use 0ms to simulate half the round-trip time between two computers on the same LAN. We use publicly available network latencies measured among 1740 computers distributed around the world [7] to simulate latencies between two computers on different LANs.

4.2 Simulations

Using these parameter values, we simulated the local and remote response times for all of the policies for both centralized and replicated architectures with unicast and multicast. Of all of the existing multicast algorithms, we know of only one that that considers the time the users' computers require for transmitting on the network in the building of such a tree, which is the HMDM algorithm [2]. We found that the cost of transmitting commands can be high in the applications we used to gather parameter values. Thus, we use HMDM to create our multicast overlays.

Our theoretical results predict that in theory, the lazy policy can significantly improve the performance of some users without significantly degrading the performance of others. To check if these improvements can be significant in practical circumstances, we consider a scenario in which a PowerPoint presentation is being given to 200 audience members around the world. Based on the published network latency data between 1740 computers [7], we set the network latencies between all users equal to those between a random subset of 200 of the 1740 computers. One issue with randomly selecting the subset is whether the subset preserves properties, such as triangle inequality and latency distributions, of the entire set. Zhang et al. [12] analyzed random subsets taken from latencies measured between 3997 computers and found that they were representative of the overall measurements. The lecturer is using a P3 desktop. Moreover, the users are organized in a centralized architecture in which the lecturer's computer is the master. Furthermore, we assume that multicast is used for communication. Finally, as mentioned before, users can notice 50ms degradations in local [11] and remote [4] response times. Thus, we set both response time degradation thresholds to 50ms.

In this scenario, the differences between the remote response times are shown in Figure 4. As Figure 4 shows, the lazy remote response times are either significantly better than (1655ms) or equal to the process-first transmission times, which agrees with the prediction made by our equations. Furthermore, the lazy remote response times can be as much as 126.6ms and 146.6ms better for some users than the transmit-first and concurrent remote response times, respectively. On the other hand, the lazy remote response times are worse by as much as 2811ms and 2721ms than those of the transmit-first and concurrent policies, respectively. These results agree with the predictions made by the equations. In addition, they show that for some users, the lazy policy provides significantly better remote response times than all other policies.

The lazy policy local response time, 700.4ms, is 50ms worse than the process-first local response times, 650.4ms, which is within the local response time degradation threshold. The transmit-first and concurrent local response times are both 873.8ms, which is significantly worse than those of the lazy policy. These results agree with the predictions made by the response time equations.

5. RELATED WORK

Our work draws on research in several diverse fields including not only collaboration architectures but also human-perception studies, distributed systems, scheduling, and mobile computing.

As mentioned above, previous work has found that users cannot distinguish between local response times below 50ms [9]. Jay et al. [4] complement these results by showing that 50ms and 25ms remote response times for visual and haptic operations, respectively, are also noticeable. In addition, they found that 50ms increments in remote response times are noticeable for both kinds of operations. While no study has directly addressed noticeable changes in local response times, one can derive them indirectly from the study of local response times by Youmans [11]. Youmans provided the participants with a button that reduced local response times by one eight and found that the participants forced these times into the 300-500ms range, which implies that 43ms decrements in local response times are noticeable.

The lazy policy relies on these thresholds to delay the processing task. The general idea of completing tasks "just in time" been studied both in single-processor and distributed [10] real-time systems. In the distributed case, end-to-end scheduling algorithms are used, which are related to the lazy policy in the multicast case. In end-to-end scheduling, a task is divided into subtasks, and each subtask is allocated to a different processor. The sub-tasks are governed by precedence constraints – subtask k cannot start until subtask k-1 completes. Distributing subtasks in this fashion is similar to building multicast overlays. Multicast schemes divide the transmission task into subtasks and schedule each subtask on a different machine. Precedence constraints are implicit as a machine must wait for a command to arrive before forwarding it. However, there are several important differences between end-to-end scheduling and the lazy policy. First, in end-to-end scheduling, the system has freedom in mapping a task to any processor, while in a collaborative system, the processing architecture constrains the mapping. In particular, regardless of the architecture, input (output) processing must be done on all master (master and slave) machines. Thus, processing tasks are not distributable in end-to-end scheduling. Second, the two kinds of systems have fundamentally different goals. The main goal of end-to-end scheduling is to complete the final subtask by the overall task deadline, while the main goal of the lazy policy is to meet the processing task deadline on as many computers as possible. As a result, our scheme trades off local and remote response times, while end-to-end scheduling schemes only guarantee that a task completes on time.

"Just in time" task completion has also been used in energy-aware end-to-end scheduling algorithms for distributed mobile systems. Seshasayee et al. [8] present an energy-aware scheduling algorithm that uses slack to complete tasks just in time while maximizing the battery life of each device. For instance, if a task completes early, they use dynamic voltage and frequency scaling to reduce the amount of power consumed by the mobile devices. As a result, the task completes later, but on time. Unlike our lazy policy, which trades-off local and remote response times, their scheme trades-off application lifetime for response times.

6. DISCUSSION

The user experience in a collaborative application suffers when response times are large. In fact, first-time users of collaboration technology may never try it again if they face intolerable response times. In this paper, we have presented a new scheduling policy that, like process-first, tries to optimize local response times but, unlike the latter, uses slack time to improve remote response times. We have shown, both through formal analysis and simulations, that it is always superior to the process-first policy as it provides 1) local response times that are as good as or unnoticeably worse than and 2) remote response times either as good as or significantly better than those of the process-first policy. Thus, lazy scheduling should be used if local response times are more important. Our results also show that neither the concurrent nor the transmit-first policy dominates the lazy policy. If all performance parameters are known, our equations make it possible to determine which one of these three policies would improve the response times for a particular user.

The analysis we have presented did not account for simultaneous user commands. Concurrent interaction does not occur in many practical collaboration scenarios. For instance, in the PowerPoint logs we analyzed, there was only one presenter who had large think times between commands. Also, in the Checkers logs we analyzed, the players used social protocol to avoid entering commands at the same time. In general, however, simultaneous commands can occur. The scheduling policies we considered are flexible enough to handle them if we make the reasonable assumption that tasks of a command are completed atomically with respect to those of other commands. A similar approach to atomicity is made in popular multi-player online games. For example, the Quake server repeatedly performs a three-step loop [1]: 1) read client commands; 2) process them; and 3) send replies to clients. The server does not check for new client commands until it processes all of the commands it read in the previous loop iteration. A similar loop executes on each client, only it receives commands from the local user and the server and sends the user's commands to the server. Such atomic processing is performed in all published gaming literature we found.

A side effect of scheduling tasks for a command atomically is that commands are not allowed to overlap, which can potentially rule out telepointer and other continuous motions such as drag and drop and, in our studied scenario of PowerPoint presentations, quickly browsing though a succession of slides and animations. Non-overlapped execution of continuous operations may result in these operations appearing discontinuous to users. However, as we illustrate next, this does not manifest itself as a real problem in a large number of scenarios when tasks for a command are scheduled atomically. Consider a telepointer motion. The motion will appear smooth if the telepointer commands are generated and processed at a rate of thirty per second (or one every 33ms). We have found that on a P4 desktop, the processing and transmission times of a telepointer command are 0.83 and 0.06ms, respectively. Thus, if the application generates a telepointer command every 33ms, then a P4 desktop has 32.17ms to perform the transmission task. When unicast is used, the command can be transmitted to as many as 536 destinations. When multicast is used, the number of users supported is even higher because each P4 computer that forwards commands can forward to as many as 536 destinations. The telepointer costs, as well as costs of other continuous motions (such as a drag-and-drop), are going to be different on other processors. Although we expect that they are still low, studies are needed to determine these costs and the range of scenarios in which the scheduling policies we considered can support them.

Another issue with simultaneous commands is that they may conflict when different users enter them. In collaborative systems, conflicts are resolved by concurrency control mechanisms, which have their own processing and transmission tasks. Scheduling of these tasks is beyond the scope of this paper. Regardless of the how exactly the concurrency control tasks are scheduled, which is left as future work, their impact on response times can be captured by the equations we have presented. Concurrency control mechanisms are pessimistic or optimistic by nature. Pessimistic schemes prevent the execution of a user's command until they verify that the command does not conflict with other users' commands. Our equations can handle the time required for such verification by adding it to the processing time of the command. Optimistic schemes do not prevent an action from executing and they recover from any conflicts that result using transformation operations, state rollback, or undo/redo mechanisms [3]. Our equations handle the time required to use these mechanisms by adding it to the processing time of the command which caused the conflict. Thus, our equations support concurrency control tasks though they do not consider efficient scheduling of these tasks.

When conflicts occur, the effect of pessimistic and optimistic concurrency control mechanisms on the response times is not clear. The reason is that in both cases, processing times of some commands will be increased causing new tradeoffs between local and remote response times. For instance, when conflicts are resolved using operation transformations, remote response times are improved but can significantly degrade local response times [6]. On the other hand, when there are no conflicts, then only pessimistic schemes inflate processing times. Thus, when lazy scheduling is used, optimistic schemes allow for larger processing delays than pessimistic schemes. This is yet another performance argument for using optimistic instead of pessimistic schemes.

Interestingly, Greenberg and Marwood [3] observe that the amount of conflicts that happen under optimistic concurrency control mechanisms is a function of remote response times. More specifically, the lower the remote response times, the sooner a user becomes aware of other users' actions, and thus makes fewer conflicting actions. Our equations can be used to choose a scheduling policy that provides the lowest remote response times among inputting users (as opposed to observing users), and thus reduce potential conflicts. A reduction in conflicts would reduce the use of conflict recovery mechanisms. In turn, this enables the lazy scheduling policy to further delay processing commands, thus improving remote response times even more in a self-feeding cycle. This cycle calls for reducing remote response times as much as possible, even if the users cannot notice the reduction.

Our work can be extended in a number of ways. A useful notion would be to combine the idea of lazy scheduling with concurrency control. Merging these ideas can create a new pessimistic-optimistic hybrid concurrency control scheme, which works as follows. When a user enters a command, the scheme behaves pessimistically at first. If the time required to check for conflicts is longer than the local response time degradation threshold, however, the scheme turns optimistic and uses conflict resolution mechanisms to handle any conflicts that occur. If, on the other hand, the time required for conflict checking is less than the local response time threshold, the scheme remains pessimistic. Such a scheme would be useful because the user would never notice the delays during the pessimistic phase, yet conflicts detected during the phase can be handled easily by rejecting the user's command.

It would also be useful to study the impact on performance of different response time degradation thresholds. As mentioned above, we used 50ms as the noticeable response time thresholds in our PowerPoint simulations. However, the studies that found these thresholds did not use PowerPoint commands. Although unlikely, the noticeable response time differences in PowerPoint may be more than 50ms, and user studies are needed to verify this claim. Previous work has shown that tolerable response time thresholds can be higher than their noticeable counterparts [9]. We used noticeable thresholds in our simulations based on the fact users would always prefer a system that provides noticeably better performance, if all else is equal (e.g., cost, functionality, etc.). Moreover, the tolerable thresholds are a function of users' expectations and can be expected to drop as users find systems with better than expected performance.

Although we have focused on scheduling tasks on a single-core, the policies we have presented can be ported to multi-core systems. The lazy policy can be adapted to multi-core scenarios by delaying the processing task on all of the (one of the) cores if the processing task is (is not) parallelizable. Moreover, the concurrent scheme can perform the processing and transmission tasks in parallel on different cores. The sequential schemes would simply use all cores for one and then for the other task. Future studies are needed to evaluate the response time differences provided by these policies on multi-core systems.

It would also be useful to create a collaborative framework that periodically evaluates the equations we have provided during collaborative sessions, and based on the history of past collaborations of the same kind, dynamically chooses the policy that best suits the current needs of the collaborators. Moreover, it would be useful to investigate the dual of the lazy (process-first) policy presented here, one in which the transmission task is given precedence over the processing task. It would also be useful to tie multicast to scheduling by building an overlay that accounts for the scheduling policy. Finally, it would also be useful to compare the scheduling policies used in current commercial collaborative systems, such as those used in multi-player games, with the lazy policy. Some of these applications, such as Quake, can have large processing and transmission costs. The lazy policy should do well in these applications because of the additive effect on the remote response time improvement when processing costs are high.

7. ACKNOWLEDGEMENTS

This work was funded in part by a Natural Science and Engineering Research Council of Canada scholarship, a Microsoft Research fellowship, and NSF grants IIS 0312328, IIS 0712794, and IIS-0810861. The comments of the reviewers substantially improved the paper.

8. REFERENCES

[1] Abdelkhalek, A., Bilas, A., and Moshovos, A. 2001. Behavior and performance of interactive multi-player game servers. ISPASS 2001.

[2] Brosh, E. and Shavitt, Y. 2004. Approximation and heuristic algorithms for minimum delay application-layer multicast trees. INFOCOM 2004.

[3] Greenberg, S. and Marwood, D. 1994. Real time groupware as a distributed system: concurrency control and its effect on the interface. CSCW 1994.

[4] Jay, C., Glencross, M., and Hubbold, R. 2007. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. TOCHI 2007.

[5] Junuzovic, S. and Dewan, P. 2008. Serial vs. concurrent scheduling of transmission and processing tasks in collaborative systems. CollaborateCom 2008.

[6] Li, D., and Li, R. 2008. An operational transformation algorithm and performance evaluation. JCSCW, 17, 5-6.

[7] p2pSim: a simulator for peer-to-peer protocols. http://pdos.csail.mit.edu/p2psim/kingdata. Mar 4, 2009.

[8] Seshasayee, B., Nathuji, R., and Schwan, K. 2007. Energy-aware mobile service overlays: cooperative dynamic power management in distributed mobile systems. Autonomic Computing (ICAC) 2007.

[9] Shneiderman, B. 1998. Response time and display rate. in Designing the user-interface: strategies for effective human-computer interaction. Addison-Wesley Longman.

[10] Sun, J. and Liu, J. 1996. Synchronization protocols in distributed real-time systems. ICDCS 1996.

[11] Youmans, D.M. 1981. User requirements for future office workstations with emphasis on preferred response times. IBM United Kingdom Laboratories.

[12] Zhang, B., Ng, T.S.E, Nandi, A., Riedi, R., Druschel, P., and Wang, G. 2006. Measurement-based analysis, modeling, and synthesis of the internet delay space. IMC 2006.