Serial vs. Concurrent Scheduling of Transmission and Processing Tasks in Collaborative Systems

Sasa Junuzovic and Prasun Dewan

Department of Computer Science, University of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA
{sasa,dewan}@cs.unc.edu

Abstract. In collaboration architectures, a computer must perform both processing and transmission tasks. Intuitively, it seems that these independent tasks should be executed in concurrent threads. We show that when multiple cores are not available to schedule these tasks, a sequential scheme in which the processing (transmission) task is done first tends to optimize feedback (feedthrough) times for most users. The concurrent policy gives feedback and feedthrough times that are in between the ones supported by the sequential policies. However, in comparison to the process-first policy, it can noticeably degrade feedback times, and in comparison to the transmit-first policy, it can noticeably degrade feedthrough times without noticeably improving feedback times. We present definitions, examples, and simulations that explain and compare these three scheduling schemes for centralized and replicated collaboration architectures using both unicast and multicast communication.

Keywords: collaboration architecture, scheduling policy, response time, feedback time, feedthrough time, unicast, multicast, simulations.

1 Introduction

An important issue in collaborative systems is the architecture of the implementation, which has an impact on the performance, the level of sharing, and correctness of the system. This area has been studied extensively [3] and has identified several important dimensions. In this paper, we focus on two related questions that have been largely ignored previously – the manner in which the tasks needed to implement collaborative systems are scheduled and the impact of the scheduling policy on local and remote response times. We refer to local response times as *feedback times* and remote response times as *feedthrough times*. Feedback times are also sometimes called simply response times [4].

Two mandatory tasks performed by a collaborative system are processing and transmission of user commands. The nature of these tasks depends on (a) whether computation is centralized or replicated and (b) whether the commands are unicast or multicast. We consider all four cases in the evaluation of policies for scheduling these tasks.

The implementation and evaluation of scheduling schemes depend on how many cores are available for scheduling. For example, if two cores are available for

E. Bertino and J.B.D. Joshi (Eds.): CollaborateCom 2008, LNICST 10, pp. 746–759, 2009.

scheduling, it is possible to carry out processing and transmission tasks in parallel. Thus, additional cores have the potential to improve feedback and feedthrough times. However, we assume only one core is available to execute the tasks of a collaborative application, leaving multi-core scheduling as future work.

The rest of this paper is organized as follows. We first describe more precisely the processing and transmission tasks. We then motivate, illustrate, and qualitatively compare the sequential and concurrent policies for scheduling these tasks. Following this, we present simulation results that quantitatively compare these policies in realistic collaborations and give brief conclusions and directions for future work.

2 Processing and Transmission Tasks

The processing and transmission tasks in collaborative systems depend on the underlying architecture. Two popular collaboration architectures are the centralized and replicated architectures. In both cases, it is assumed that an application is logically separated into a program and user-interface components. The program component manages the object that is shared by all of the users. The user-interface component allows interaction with the shared object by manipulating state that is not shared by the users. A separate user-interface component runs on each user's machine.

In the centralized architecture, all of the user-interface components are mapped to a single program component. The computer running the program component is called a *master* and the other computers are called *slaves*. A master computer receives input commands from and sends output commands to all of its slaves. In addition, a master is responsible for processing all input commands and their outputs. A slave, on the other hand, is responsible for transmitting input commands from its user to the master and processing the output of all input commands. A centralized architecture with six users in which $user_1$ is the master is shown in Fig. 1 (top). The figure shows the transmission of an output for an input entered by $user_1$. In the replicated architecture, each user-interface component is mapped to the program component running on the local computer. Thus, all of the computers are masters. To keep the program components on different masters in sync, whenever a master receives an input command from the local user, it transmits the command to all of the other computers. A replicated architecture with six users is shown in Fig. 2 (top). The figure shows the transmission by $user_1$'s computer to all of the other computers after $user_1$ enters an input command.

One issue with the traditional architectures is that if inputs or outputs are large and the number of users is high, then the cost of transmitting an input or output to many users is also high. As a result, master computers can become performance bottlenecks. It is possible to overcome this problem by using the bi-architecture model [4], in which a collaborative system is separated into two sub-architectures. As in the traditional architecture case, the user-interface components are still mapped to program components; however, the mapping in this case is not bi-directional. In particular, a slave computer sends input commands to the master computer to which it is mapped, but the master computer does not have to directly send input and output commands to all of the other masters and its slaves, respectively. Instead, multicast is used allowing more than just the master to transmit the commands.

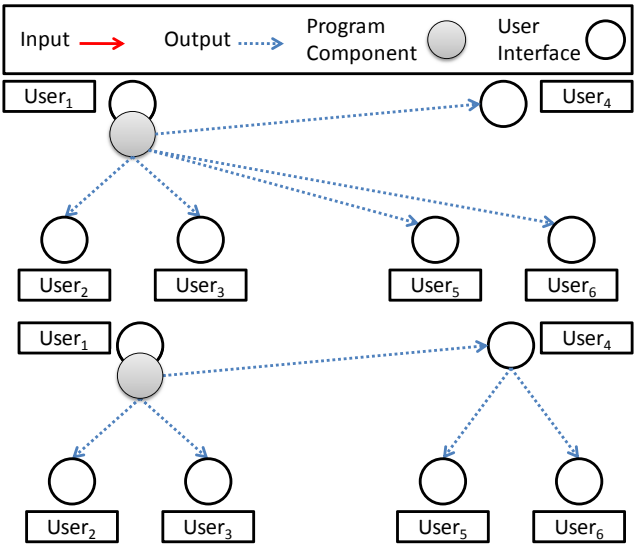

Fig. 1. (top) Traditional centralized architecture and (bottom) the bi-architecture model with a centralized architecture in which multicast is used for communication

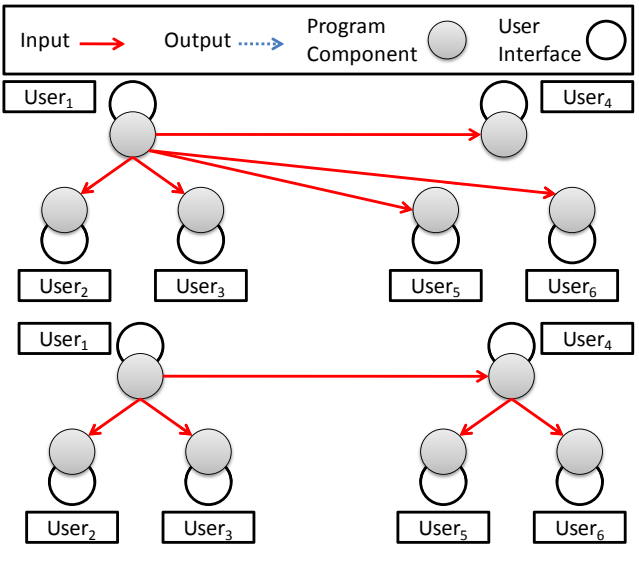

Fig. 2. (top) Traditional replicated architecture and (bottom) the bi-architecture model with a replicated architecture in which multicast is used for communication

The idea of multicast requires, for each source of messages, the construction of a multicast overlay that defines the paths a message takes to reach the destinations. The bi-architecture model makes several assumptions regarding multicast. First, because

IP-multicast is not widely deployed, the model assumes an application-layer multicast in which end-hosts form the overlay. Second, the model assumes that only the users' computers can be used in the overlay. This is consistent with the notion of peer-to-peer sharing systems.

With multicast, every computer may perform some part of the transmission task. For example, if multicast is used in the centralized architecture, then a slave computer, in addition to processing any outputs that it receives, may also need to forward the outputs to other slaves as shown in Fig. 1 (bottom). Fig. 1 (bottom) shows the transmission after $user_1$'s computer, which is the master, computes the output for a command entered by $user_1$. The master transmits the output only to computers belonging to $user_2$, $user_3$, and $user_4$. $User_4$'s computer, which is a slave, then forwards the output to the computers belonging to $user_5$ and $user_6$. Similarly, if multicast is used in the replicated architecture, a master computer that receives an input command from another master may, in addition to processing the command, have to forward it to other masters as shown in Fig. 2 (bottom). Fig. 2 (bottom) shows the transmission of an input command entered by $user_1$. $User_1$'s computer transmits the command only to computers belonging to $user_2$, $user_3$, and $user_4$. $User_4$'s computer forwards the command entered by $user_1$ to computers belonging to $user_5$ and $user_6$. When unicast is used for communication among the computers, the bi-architecture model reduces to the traditional model.

3 Scheduling of Tasks

While the bi-architecture model specifies the tasks that the users' computers will carry out, it leaves as an implementation issue the scheduling of these tasks on each computer. In this section, we motivate, illustrate, and qualitatively analyze three useful scheduling policies.

3.1 Running Example

To illustrate and compare the policies we consider in this paper, we will use the replicated-multicast architecture shown in Fig. 2 (bottom) with the following additional properties: (a) $user_1$'s computer transmits commands first to $user_4$, then to $user_2$, and finally to $user_3$, while $user_4$'s computer forwards the commands first to $user_5$ and then to $user_6$; (b) $user_1$ enters all of the commands; (c) the users all have the same computers; (d) the network latency between any two computers is D; (e) the time the computers require to process an input and output command is 3T and T, respectively; and (f) the time the computers require to transmit an input command to a single destination is T. The relationships between the various times were carefully selected to allow this theoretical example to be used to easily compare all of the policies. In our simulations, we use realistic values for all of these parameters, which do not assume, for instance, that the network latencies among the users are the same.

For all of the scheduling policies we consider, we illustrate $user_1$'s feedback time and $user_6$'s feedthrough time. The reason we consider $user_6$ instead of other users is because $user_6$ is the "farther" from the source than any other user. As Fig. 2 (bottom)

shows, the path from $user_1$ to $user_6$ is longer than the path from $user_1$ to any other user, except $user_5$. The paths from $user_1$ to $user_5$ and $user_6$ both go through $user_4$. Since $user_4$ transmits first to $user_5$ and then to $user_6$, we consider $user_6$ to be farther away than $user_5$ is from $user_1$. Once the calculation of $user_6$'s feedthrough time is understood, the feedthrough times of other users are easy to derive. Therefore, these feedthrough times are presented without derivation in Table 1.

3.2 Process-first and Transmit-First Scheduling Policies

One way of scheduling the processing and transmission tasks it to execute them sequentially. There are two sequential policies possible in which either the processing or the transmission task is performed first.

The process-first policy provides better feedback times than the transmit-first policy because, unlike the transmit-first policy, it does not delay the processing of a command until the transmission task completes. Comparing the feedthrough times of the two policies is more complicated. Transmitting first from a source seems to improve the feedthrough times of the destinations. However, as each destination may also be a source, delaying the processing of the received command can increase the feedthrough time seen by the local user.

To understand the influence of these factors on the relative feedthrough performance of the two policies, consider the feedthrough time of $user_6$ in our running example. In all policies, this time consists of four components: (1) the total network delay the command experiences, (2) the time taken by $user_6$'s computer to process the command, (3) $user_1$'s delay, and (4) $user_4$'s delay, where $user_1$'s ($user_4$'s) delay is equal to the time that elapses from the moment $user_1$'s ($user_4$'s) computer receives a message to the moment it transmits it to $user_4$'s ($user_6$'s) computer. The first two components have the same values in all policies. A command always traverses the network twice, which requires 2D time. Since $user_6$'s computer does not transmit commands to other computers, once it receives the command, it always processes the command and the corresponding output in 4T time. The values of the other two components are policy-specific.

The calculation of the policy-specific components when process-first and transmit-first scheduling are used is shown in Fig. 3 (top) and Fig. 3 (bottom), respectively. As Fig. 3 (top) shows, with the process-first policy, $user_1$'s delay is equal to the time $user_1$'s computer requires to process the input command and the corresponding output, 4T, plus the time it takes to transmit the input to a single destination, T. Thus, $user_1$'s delay is equal to 5T. As Fig. 3 (top) also shows $user_4$'s delay is equal to the time $user_4$'s computer requires to process the input command and the corresponding output, 4T, plus the time it takes to transmit the input to two destinations, 2T. Thus, $user_4$'s delay is equal to 6T. Hence, $user_6$'s feedthrough time with the process-first policy is 4T+2D+5T+6T=15T+2D. On the other hand, as Fig. 3 (bottom) shows, when transmit-first scheduling is used, $user_1$'s delay is equal to the time $user_1$'s computer requires to transmit the input command to a single destination, T, while $user_4$'s delay is equal to the time $user_4$'s computer requires to transmit the command to two destinations, 2T. Hence, $user_6$'s feedthrough time is 4T+2D+T+2T=7T+2D. The feedthrough times for the remaining users are given in Table 1.

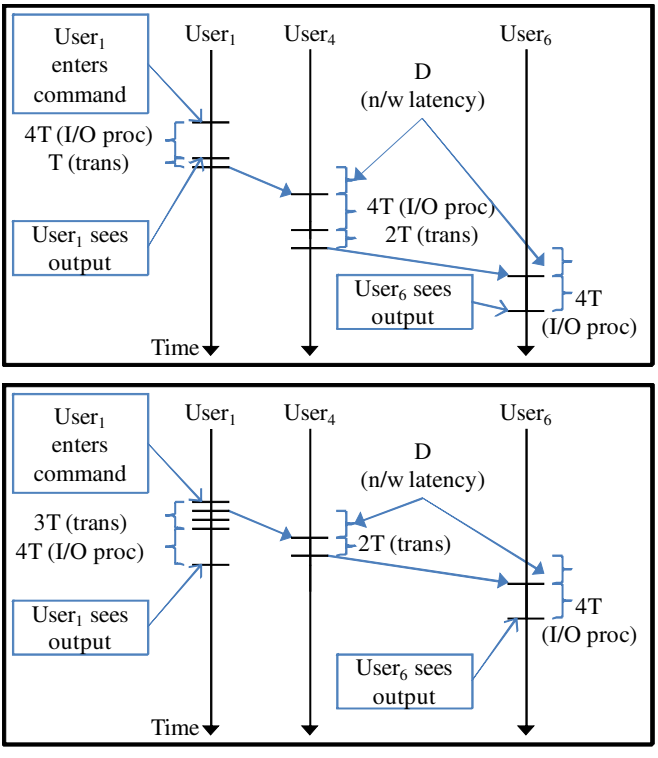

Fig. 3. User$_1$'s feedback time and user$_6$'s feedthrough time for the architecture in Fig. 2 (bottom) when the (top) process-first and (bottom) transmit-first scheduling is used

Table 1 shows that *in this theoretical example* the transmit-first policy gives better feedthrough times than the process-first policy for all of the users. However, this is not true in all cases. For instance, suppose there were five more users in our example, and the multicast overlay was organized so that these five users all receive inputs from user$_4$'s computer. In this case, user$_4$'s computer would still receive the command 4T earlier with the transmit-first than with the process-first policy but would have to transmit for 5T longer before processing it. Hence, the benefit from receiving input early can, theoretically, get outweighed by the transmission cost; in this example, user$_4$'s feedthrough time would increase by T. Such an increase can only happen when multicast is used. However, in our experience with a state of the art multicast scheme, such an increase does not really occur because usually a small number of computers actually forward commands. Moreover, an even smaller number of computers forward commands to many destinations. As a result, the number of destinations a computer forwards to is usually small enough that the total transmission cost for a node is smaller than the benefit the node receives when the transmit-first policy is used. Hence, we expect that the transmit-first policy will provide better feedthrough times than the process-first policy to most, if not all, of the users.

Our running example also shows that the process-first policy gives better feedback times than the transmit-first policy. As Fig. 3 (top) shows, user$_1$'s process-first

Table 1. $User_1$'s feedback times and $user_2$'s, $user_3$'s, $user_4$'s, $user_5$'s, and $user_6$'s feedthrough times under the three scheduling policies

Policy	Process-first	Transmit-first	Concurrent
$User_1$	4T	7T	7T
$User_2$	10T+D	6T+D	8T+D
$User_3$	11T+D	7T+D	10T+D
$User_4$	9T+D	7T+D	8T+D
$User_5$	14T+2D	6T+2D	8T+2D
$User_6$	15T+2D	7T+2D	10T+2D

feedback time is 4T. On the other hand, as Fig. 3 (bottom) shows, $user_1$'s transmit-first feedback time is 7T.

In summary, a sequential scheme in which the processing (transmission) task is done first tends to optimize feedback (feedthrough) times for most users If we are interested in both good feedback and good feedthrough times, it is attractive to investigate a concurrent approach in which separate threads perform the processing and transmission tasks.

3.3 Concurrent Scheduling Policy

Intuitively, we would expect a concurrent policy to give feedback and feedthrough times in between those supported by the two sequential policies. In fact, in this policy, it is possible to get feedback times that are as bad as those of the transmit-first policy and feedthrough times that are as bad as those of the process-first policy.

Let us analyze what happens on $user_1$'s computer in our running example when the computer receives an input command. As described above, in this case, the processing and transmission task require 4T and 3T time, respectively. 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. Finally, for illustration purposes, we assume here that the length of the scheduling quantum is much less than the processing and transmission costs. In our simulations, we in fact, use a much more realistic value of 10ms for the quantum size. Given these assumptions and our earlier assumption that a single core is available for scheduling, the execution of these tasks for the concurrent and the two sequential policies is illustrated in Fig. 4. As Fig. 4 shows, with the

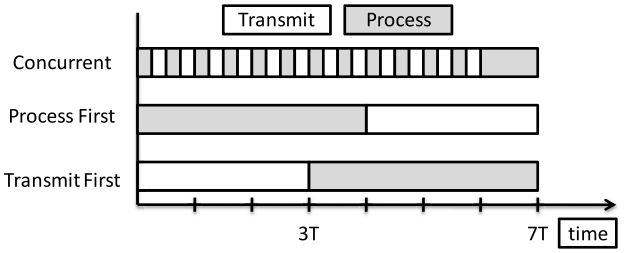

Fig. 4. Process and transmission task completion times for user$_1$'s computer for the concurrent, process-first, and transmit-first scheduling policies

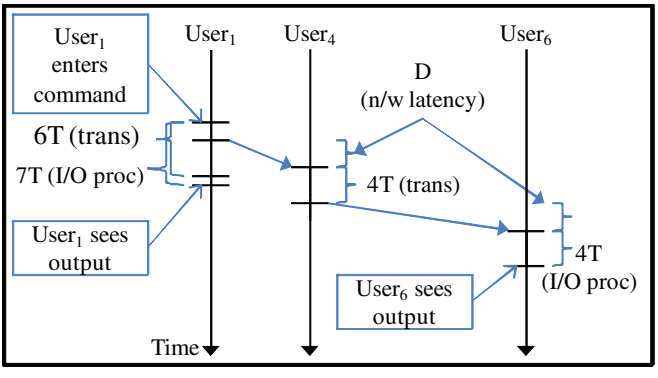

Fig. 5. User$_1$'s feedback time and user$_5$'s feedthrough time for the multicast communication architecture in Fig. 2 (bottom) when the concurrent scheduling is used

concurrent policy, the shorter transmission task completes in 6T time, which is twice the time it takes to complete when the task runs standalone. As Fig. 4 also shows, with the concurrent policy, the longer processing task completes in 7T time, which is equal to the total time required to process the processing and transmission tasks sequentially. We can generalize the figure as follows: when the processing and transmission tasks are executed concurrently, (a) the shorter of the two will complete in exactly twice the time it would complete were it running standalone, and (b) the longer of the two will complete in exactly the time required to run the two tasks sequentially. In this example, the processing task is the longer one, so user$_1$'s feedback time is 3T+4T=7T.

As mentioned earlier, user$_6$'s feedthrough time equals 4T+2D + user$_1$'s and user$_4$'s delays. As Fig. 5 shows, user$_1$'s delay with the concurrent policy is equal to the time user$_1$'s computer requires to transmit the command to a single destination while concurrently processing the command. Since transmitting to a single destination takes T time and the processing task takes 4T time, the transmission to a single destination completes in 2T time since it is the shorter of the two tasks. Similarly, as Fig. 5 also shows, user$_4$'s delay is going to be 4T. Thus, user$_6$'s feedthrough time equals 4T+2D+2T+4T=10T+2D. The feedthrough times of all users are shown in Table 1.

Based on the feedback times and the feedthrough times in Table 1, it seems that *in this theoretical example* the concurrent policy combines the worst of both sequential policies as its feedback time is no better and its feedthrough times are worse than the transmit policy. Of course, it is easy to change the example to ensure that the concurrent policy offers feedback and feedthrough times between those of the transmit-first and process-first policies. Here we chose the example to make the subtle point that this is not always the case. In general, however, if the goal is to equally favor feedback and feedthrough times, the concurrent policy should be used.

3.4 Simultaneous Commands

One issue we have not addressed so far is the scheduling of multiple simultaneous commands. In general, two types of commands can occur concurrently with $user_1$'s input command: 1) another collaboration-unaware user input command, or 2) a collaboration-aware command, such as one caused by the concurrency control or awareness mechanisms. Collaboration-aware commands have their own processing and/or transmission tasks that must be scheduled. Scheduling of these commands is beyond the scope of this paper and we leave it as important future work. In this paper, we make the reasonable assumption that tasks for a command are completed atomically with respect to tasks for other commands. Given this assumption, once a computer begins to perform tasks for $user_1$'s input command, other commands cannot affect the feedback and feedthrough times of the command. However, it is possible that when $user_1$'s input arrives at a computer, the computer performs tasks for several other commands before beginning the tasks for this command. The time the computer takes to complete the tasks for these other commands adds to the feedback and feedthrough times of $user_1$'s command.

Collaboration-aware commands simply add some time to the feedback and feedthrough times that is independent of the user command scheduling policies. Hence, the differences in feedback and feedthrough times illustrated above stand. User input commands also add some time to the feedback and feedthrough times of $user_1$'s command that is independent of our choice of scheduling policy. The reason is that regardless of the scheduling policy, the time that elapses from the moment a computer begins performing the first task for a user command to the moment it completes the final task for the command is the same. Consider $user_1$'s computer in our running example. The time it takes to process $user_1$'s input command and output and transmit the input command is 7T in all cases. Thus, the illustrated feedback and feedthrough time differences in our running example for the three scheduling policies again stand.

4 Simulations

Our work so far has made several conclusions about the relative performance of the three scheduling policies based on theoretical arguments. While these results are a contribution on their own, it is important to see if the differences shown through a theoretical evaluation can be significant when the policies are evaluated in practical scenarios.

We determined the performance of the scheduling policies in practical scenarios using bookkeeping or accounting mathematical equations that simulate a collaborative system. Such simulation approaches are popular in other fields such as networking and real-time systems. Because of lack of space, we omit the equation details.

4.1 Parameter Values

To perform meaningful simulations we need realistic values for the parameters that influence the performance of the three 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 three different applications, but as we have space to talk about the results with only one these applications, we focus only on it.

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 presentations.

To obtain the processing and transmission time parameter values, we created a collaborative session with several computers. 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 processing and 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 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 transmission times 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. Similarly, based on pings done between computers on different LANs, we use 15ms and 177ms to simulate half the round-trip time between a Northwest and a Southwest U.S. LAN and an East-coast U.S. and an Indian LAN, respectively. These values defined the minimum and maximum network latencies in our evaluation.

4.2 Simulations

Using these parameter values, we simulated the feedback and feedthrough times for all of the policies for both centralized and replicated architectures when unicast and multicast are used for communication. 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]. In our experience, the cost of transmitting commands can be high in data-centric applications such as PowerPoint. Thus, we implemented HMDM in Java and used it to create our multicast overlays.

4.3 Process-First vs. Transmit-First

Our theoretical results predict that the process-first policy gives better feedback times but worse feedthrough times than the transmit-first policy, and vice versa. To check if this difference 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 ping times we reported earlier, we assume that the latencies between all of the users are between 15ms and 177ms. The lecturer is using a next generation PDA device. Moreover, the users are organized in a centralized architecture in which the lecturer's computer is the master. Finally, we assume that multicast is used for communication.

Previous work has shown [6] that users can notice feedback times greater than 50ms. We consider a 50ms increment in feedback times significant. Moreover, since we know of no feedthrough thresholds, we assume that 50ms increments in feedthrough times are also significant. In this scenario, the process-first policy feedback time, 650.4ms, is significantly better than the transmit-first feedback time is, 761.2ms. The difference between the process-first and transmit-first feedthrough times are shown in Fig. 6. As Fig. 6 shows, the process-first feedthrough times results are significantly worse, by as much as 2804ms. Hence, there are cases when the process-first policy can provide significantly better feedback times and significantly worse feedthrough times than the transmit-first policy. The results of another simulation, which we do not have room to present, show that the process-first feedthrough times can be significantly better than the transmit-first feedthrough times. However, for a large majority of the users (99%), the feedthrough times were actually either noticeably lower or not noticeably higher with the transmit-first than the process-first policy.

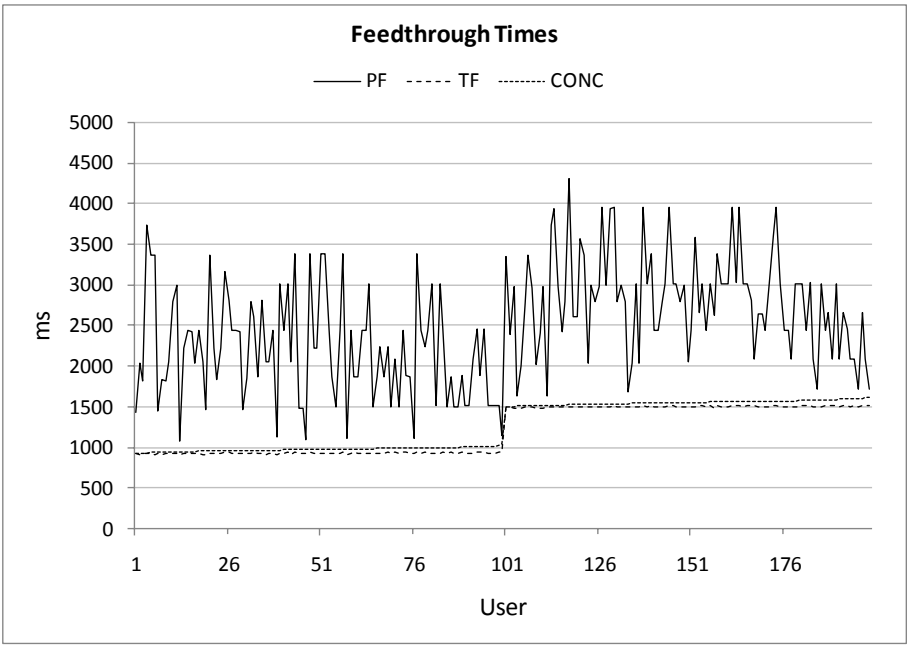

Fig. 6. Feedthrough times for the process-first (PF), transmit-first (PF), and concurrent (CC) scheduling policies

4.4 Concurrent vs. Sequential

Our second theoretical result was somewhat counter-intuitive. It showed the concurrent policy can be as bad as the transmit-first policy in terms of feedback times and worse than the transmit-first policy in terms of feedthrough times. To find out if the feedthrough time differences can be significant, we consider the same scenario as in the previous result.

The scenario simulation results confirm that the concurrent policy feedback times can be the same as those of the transmit-first policy (761.2ms for both). Moreover, the simulation feedthrough times are shown in Fig. 6 and they show that the concurrent policy feedthrough times can be significantly worse, by as much as 110.0ms, than the transmit-first feedthrough times. Even worse is the fact that more than one quarter of the users experience these significant feedthrough time degradations.

Another theoretical result regarding the concurrent scheduling policy is that it is useful if both feedback and feedthrough times are equally favored because with the concurrent policy, these times can be in between those provided by the process-first and transmit-first scheduling policies. It turns out that these differences can be significant in the following practical scenario.

Consider again the PowerPoint scenario described earlier with three differences: (a) there are only 100 users watching the presentation, (b) they are all in the same LAN as the lecturer and thus experience only LAN network latencies (i.e. 0ms), and (c) unicast is used for communication.

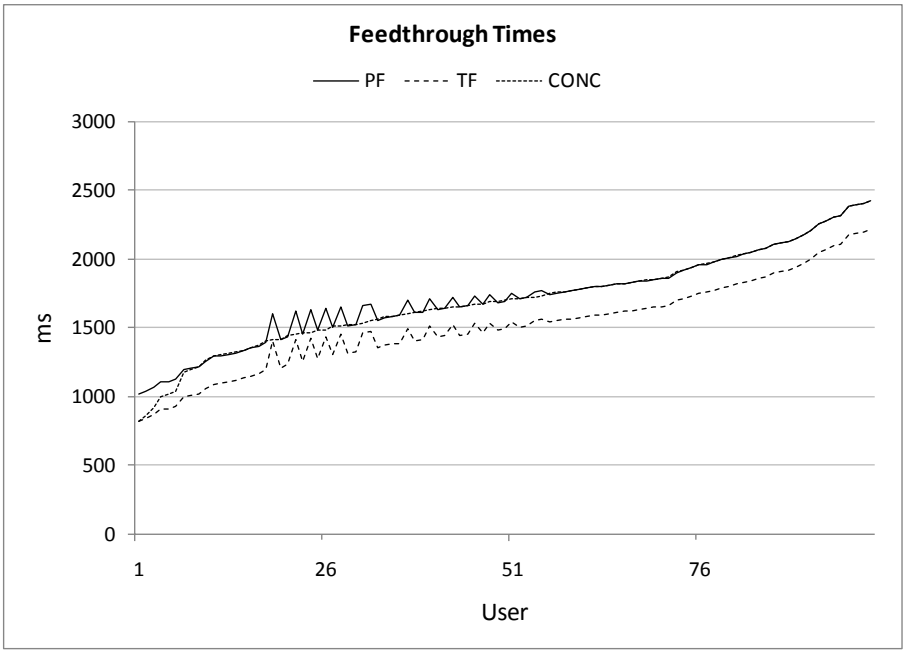

Fig. 7. Feedthrough times for the process-first (PF), transmit-first (PF), and concurrent (CC) scheduling policies

In this case, the concurrent policy feedback time, 860.4ms, is significantly worse than the process-first policy feedback time, 650.4ms, but is significantly better than the transmit-first feedback time, 1502.4ms. Moreover, the concurrent policy feedthrough times are significantly better than those of the process-first policy for some users. In addition, for those same users, the concurrent policy feedthrough times are significantly worse than those of the transmit-first policy, as shown in Fig. 7. As Fig. 7 shows, the feedback time for user 6 is (a) 105.2ms better with the concurrent than with the process-first policy and (b) 100.0ms worse with the concurrent than with the transmit-first policy.

5 Conclusions and Future Work

We show that scheduling of processing and transmission tasks can significantly influence interactivity. As these are independent tasks, intuitively, it seems that they should be executed in concurrent threads scheduled by the operating system. However, we show that, when a single-core is available for processing, this policy is dominated in several realistic collaborations by sequential policies that are aware of the nature of these two tasks. This result also has an implication for multi-core scheduling systems. These systems tend to require an application to decompose its processing into one or more concurrent threads and schedule these threads on as many physical cores/processors as available. Our results show that when the processing and

transmission tasks cannot be scheduled simultaneously on multiple cores/processors, it may be better, in many scenarios, to execute them in a single thread using process-first or transmission-first scheduling rather than in multiple threads. Thus the main conclusion of our work is that a generic collaboration infrastructure must support all three scheduling policies and allow them to be dynamically switched based on system and task parameters.

Certain collaborative applications adapt the amount of processing work done to ensure tolerable feedback times. For example, certain game playing applications [1] adapt the level of detail presented based on the scene and processing power of the computer. Moreover, in many applications, several independent tasks can be performed in the processing phase, and in multicast, sends and receives can be performed in different threads [5]. Therefore, it would be useful to consider new scheduling policies that take into account the fact that the processing/communication task can be adapted and broken into independent work units. It would also be useful to study the (potentially application-specific) scheduling policies used in current commercial collaborative systems, which we have not been able to determine so far. Future work is also needed to consider concurrent scheduling on multiple cores, better and more formally characterize scenarios in which various scheduling policies should be used, create an infrastructure that automatically adapts the policy based on the various system and task parameters identified here, and most importantly, study how the feedback/feedthrough tradeoff should be made in different collaborations.

Acknowledgements

This research was funded in part by a Natural Science and Engineering Research Council of Canada scholarship, a Microsoft Research fellowship, and NSF grants ANI 0229998, IIS 0312328, IIS 0712794, and IIS-0810861.

References

1. Brockington, M.: Level-of-detail AI for a large role-playing game. AI Game Programming Wisdom, Charles River Media (2002)
2. Brosh, E., Shavitt, Y.: Approximation and heuristic algorithms for minimum delay application-layer multicast trees. In: INFOCOM (2004)
3. Dewan, P.: Architectures for collaborative applications. Trends in Software Computer Supported Co-operative Work 7 (1998)
4. Junuzovic, S., Dewan, P.: Multicasting in groupware? CollaborateCom (2007)
5. Ostrowski, K., Birman, K.: Implementing High Performance Multicast in a Managed Environment. Technical Report. Cornell University (2007)
6. Shneiderman, B.: Response time and display rate. Designing the User-interface: Strategies for Effective Human-computer Interaction. Addison-Wesley, Reading (2004)