On Improving Change Management Process for Enterprise IT Services

Xiang Luo, Koushik Kar
Rensselaer Polytechnic Institute
Troy, NY 12180, USA
{luox3,kark}@rpi.edu

Sambit Sahu, Prashant Pradhan, Anees Shaikh
IBM TJ Watson Research
Hawthorne, NY 10532, USA
{sambits,prashant,aashaikh}@us.ibm.edu

Abstract

In this paper, we consider the Change Management Process for Enterprise IT services with the goal of improving the efficiency of this process, i.e., minimizing change completion time and maximizing the "change capacity". The change management process handles critical updates in the system that must be implemented by a set of executing personnel. In presence of such timing constraints and scheduling conflicts between change classes, we argue that addressing the application change management question optimally involves computing solutions to NP-hard problems. Using appropriate simulation models, we evaluate various batched scheduling alternatives to understand the effect of simple process re-engineering on the change completion time, and the change capacity of the system. In addition, we examine the benefit of executor cross-training and degree of conflicts on the performance of the system. Our results indicate that a simple longest queue based scheduling approach works well in a wide range of practical scenarios. Based on these results, we recommend process re-engineering based on longest queue based scheduling with a small degree of executor cross-training.

1. Introduction

Today a typical Enterprise has a sizeable IT infrastructure that it relies on for day to day business operations such as inventory control, billing, HR services and computing needs, and expects such an infrastructure to be available on a 24/7 basis. The size of typical IT infrastructure may vary from a few hundred servers connected through a LAN to as complex as hundreds of thousands of specialized servers connected through a high-end network distributed all over the world. The management of such an IT infrastructure based service is a very complex task that requires constant upgrades, problem diagnosis, and installation of new services. Most often an Enterprise outsources its IT infrastructure management to a IT services management company for these tasks. In this outsourced IT services model, service providers use a Global Service Delivery model to provide service in a cost effective manner at a global scale utilizing cost effective labor and skills, round the clock availability based on geography, among other considerations.

In such a service delivery model, IT infrastructures are monitored and managed in a distributed fashion where (i) monitoring platform may be stationed at USA, (ii) call centers that provide interface to Enterprises may be located at Brazil and India, (iii) service delivery team may be located at distributed locations based on skill requirements. In order to manage such a complex distributed services model, most Service Providers adopt IT Infrastructure library framework known as ITIL [1] that are in place to ensure accurate, efficient services with almost no variance through a standardized set of IT processes. The typical tasks handled in such a model are Problem Management that refers to handling of problem diagnosis and root cause determination, Change Management that refers to timely implementation of updates, and Request for Service that refers to installation of new services. A service provider usually implements several hundred IT processes for each category of these services based on client requirements.

This paper examines the Change Management process using quantitative models with the objective of reducing the end-to-end change completion time and increasing the change capacity to improve the overall efficiency of this process. There are several steps involved in the change management process. The main steps consists of generating (i) timely request for changes, (ii) routing change requests to appropriate service center groups, (iii) determining the infrastructure and services that will be affected by the requested change, (iv) obtaining approval on the window of time when change can be applied, (v) applying the change procedures, (vi) testing the infrastructure and services after the change to make sure it is successful, and (vii) finally restoring the service to normal state.

Several studies have indicated that the approval step is often the bottleneck and most complex step as it has to consider several issues to determine the change window. Some

978-0-7695-3283-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCC.2008.136

of the factors that influence the determination of change window are (i) allowed downtime period for each server and application, (ii) dependency among applications, (iii) mapping of servers to applications, and (iv) availability of change executors. Given these complex, interdependent factors, often it takes several days to determine an appropriate change window. Our study models the above process under realistic assumptions and process constraints, and examines various schemes for determining when to schedule which change request with the objective of minimizing the average change completion time.

More precisely, we consider the problem of efficient management of a set of application *changes* under possible scheduling conflicts between them, and additional constraints on the change execution times. Each application change usually represents some critical updates that need to be made for a specific application, which will affect all clients using that application. An application change typically involves (software or hardware) updates on the servers used by that application. Moreover, each change must be implemented without interruption, and has an estimated time for completion, during which the application (and all other applications that use any server being updated) is unavailable to clients. Due to the important nature of the changes, scheduling of these changes must be done as soon as possible, while respecting the scheduling conflicts and timing constraints on these changes.

We formalize the change scheduling problem under very general and realistic assumptions. We also discuss the complexity of obtaining the optimal solution, and formally argue about the difficulty of computing the optimum even under significantly simpler, special case scenarios. We then evaluate the performance of the some heuristic solution approaches, and compare their performance with the optimum in specific instances of the question. We also study the effects of the degree of conflicts, executor cross-training and batching of changes on the overall performance of the system. The results indicate that in terms of the change delay as well as change capacity, the longest queue length based heuristic performs the best, and is significantly better than picking changes to be scheduled in a random manner. We observe that average change delay does not increase significantly with the degree of conflict until about 40% of the maximum value, beyond which it increases very sharply. We also observe that performance improvement with increased cross-training of personnel shows diminishing returns, and a cross-training degree of 50% appears enough to attain a performance very close to the optimal.

2. Background and Definitions

In this section, we discuss the details of the change management process we examine in this paper. We shall also define "scheduling conflicts" and "timing constraints" that we use later in the mathematical formulation of the change management question we consider.

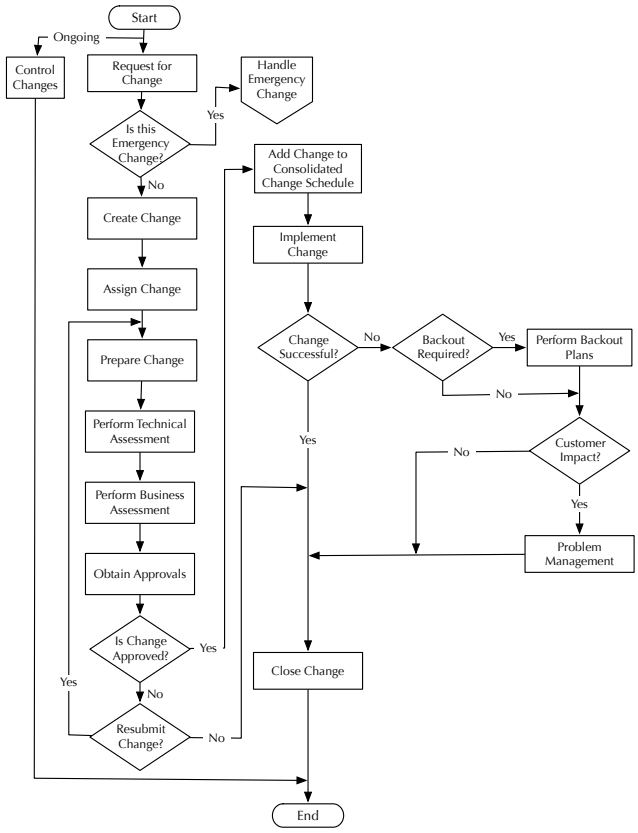

Figure 1. Change management process.

2.1. Change Management Process

Figure 1 illustrates the main steps in a typical change management process. All the steps in the process can be grouped into three main categories: change requirement determination, obtain change approval, and perform change.

Change Requirement Determination: Typically a change request is generated when there is a need to upgrade a system and/or application, probably to patch a security issue, install a more stable version of a software or upgrade to a newer version of software with additional features. There are several change determination systems that notify an application when there is such a need. The next step in the change process is to actually generate the patch bundle that is to be applied on the system or application.

Change Approval: After the patch is generated, the request is assigned to appropriate service group in a service provider organization. The next step is to obtain approval for performing this update to the system or the application.

This is an important step for an Enterprise IT system as the impacted systems will be unavailable during the time change is applied. There are stringent requirements when a server or an application may be brought down for maintenance - known as maintenance window. Also several other applications may be dependent on this server or the application. Thus one not only needs to account for maintenance window restriction for the application on which change is to be applied, one needs to also consider dependent applications' maintenance window to decide when a change can be applied. The next dependency is the availability of change executors to actually perform the change. In this change approval process, technical, business and executive factors are accounted to determine the change window. One is required to determine the schedule for the change in this process step.

Perform Change: Once the change execution schedule is determined, the change request is implemented by the assigned executor. The change executor also tests the system to make sure that it is successful and the system is stable after the effected change. If there is failure, this is reported back as a problem scenario and roll back is performed to the original state. In the failure scenario, this is reported to problem resolver group.

2.2. Scheduling Conflicts

A scheduling conflict is said to exist between two changes if they can not be scheduled at the same time. Such conflicts typically arise due to the overlap in the resources on which they are to be implemented. Applications often use a common server, or a common set of servers, thereby resulting in conflicts between the changes of those applications. As an example, consider two changes that involve a set of software updates, on the servers that implement the corresponding applications. In this case, two changes will have a conflict if they have a common server on which both changes need to be implemented. Therefore, a valid schedule of changes must be such that no two changes with conflicts are scheduled simultaneously.

2.3. Timing Constraints

As discussed above, there can be constraints on when an application can be down ("application downtime"), to minimize the effect of the application downtime on the overall client base. For example, if an application is primarily used by clients in China, we could want to implement a change for that application between 2am and 4am Beijing Time on any day. As another example, an application that is related to trading of stocks must be carried out at times and days when the stock market is closed. Note that scheduling of an application change on a set of servers not only results in the unavailability of the application during the period the change is scheduled, but also that of all other applications that use any of the servers on which the change is implemented. As a result, an application change must be scheduled at a time that corresponds to a permissible downtime of that application, as well as all other "conflicting" applications (i.e., applications which use any common servers).

In addition to the above, there can be timing constraints on an application change based on the availability of the personnel that are capable of (or responsible for) implementing the change. The person who implements (executes) a change (the change executor, or simply the *executor*) may be available only at certain times on certain days of the week. Therefore, an application change must be scheduled during the times the corresponding executor is available, while respecting the individual timing constraints of the changes (based on constraints on the application downtimes). In a more complex scenario, where a change can be implemented by one of multiple executors, the change management question involves an *assignment* question as well (i.e., assigning each change to an executor), and scheduling these changes while respecting the timing constraints of the applications and the executing personnel.

3. Formulation

3.1. System Model

Let A denote the set of applications that we consider, and let S be the set of all servers used by the applications. The set of servers used by application $i \in A$ is represented by $S_i \subseteq S$. Two applications are said to be in *conflict* with each other if they use any common server, i.e., $S_i \cap S_j \neq \phi$. Let $A_i \subset A$ denote the *conflict set* of application i, i.e., the set of applications (not including i itself) that conflict with application $i \in A$. Application change requests arrive at the change management/scheduling system according to a (typically unpredictable) random process. The goal of the change management system is to schedule these changes efficiently, in an online manner, without using any a priori knowledge of future change requests. Each change k is associated with a unique application, denoted by $A(k)$. A change k requires time d_k to be executed; although this time may not be known exactly in advance, it can be estimated, and these estimates can be used in making the scheduling decisions. Two changes k and k' are said to conflict with each other if the corresponding applications, $A(k)$ and $A(k')$, are in conflict with each other. Two conflicting changes cannot be scheduled at the same time. Since an application conflicts with itself (by default), two changes of the same application can not be scheduled together. The conflicts between applications can be conveniently modeled as a *conflict graph* $G = (V, L)$, where each vertex of the graph corresponds to an application ($|V| = |A|$), and an

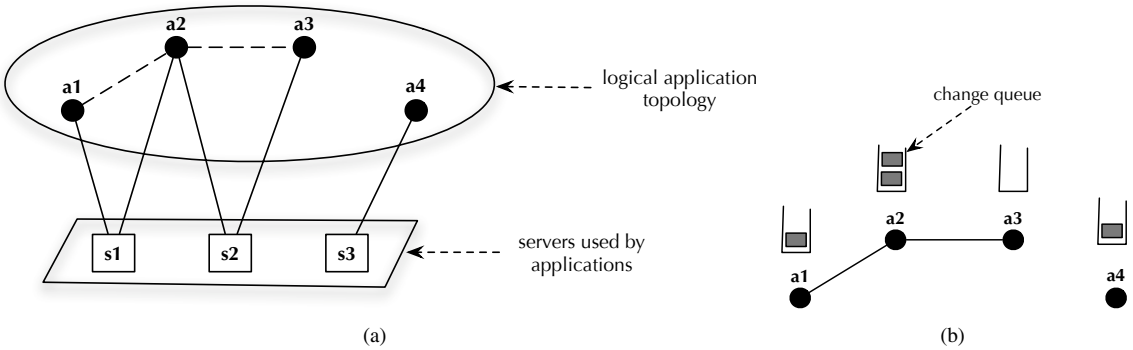

Figure 2. Application conflict graph with change queues: (a) The sets of servers used by applications $a1, a2, a3, a4$ are $\{s1\}, \{s1, s2\}, \{s2\}, \{s3\}$ respectively; the dotted lines between applications indicates the derived application conflicts; (b) The application conflict graph with application change queues.

(undirected) edge exists between two vertices in G if they conflict with each other. Clearly, a set of changes that can be scheduled at the same time must be such that their corresponding applications constitute an *independent set* in G.

The definition of the scheduling conflicts between changes in our case, as described above, can be motivated as follows. Any change implementation/execution requires updating the servers that are used by the corresponding application. While it is possible that a particular change may not require making updates on all servers that the corresponding application uses, proper testing of the application after the change implementation (to determine the correctness of the implementation), and any necessary debugging will potentially require involvement of all servers used by the application. This also explains why the notion of conflicts between changes can be aggregated in terms of the applications, as obvious from our definition of the conflict graph.

We also assume that changes of any application are implemented in an integrated (or *indivisible*) manner using the necessary servers. In other words, any one server may be used only for a part of the time over which a change is implemented, we assume that all servers in S_i are unavailable for any other purpose (including implementing other changes) during the entire period of time during which a change of application i is implemented. This assumption is motivated by practical considerations: in general, implementing a change on a set of servers, including the associated testing and debugging, requires the use of the servers in a away that cannot be calculated or predicted in advance. This implies that the entire set of servers S_i must remain "offline" for the entire duration when change of application i is being implemented. This implies that scheduling of conflicting changes can not be "pipelined" across different servers to improve efficiency. From this assumption, it follows that it is sufficient to consider the change scheduling question in the context of the *logical* application-topology instead of the *physical* server-topology. Thus, in the rest of this paper, we would consider and address the change scheduling question at this logical level (using the application-level conflict graph topology like the one shown in Figure 2), without concerning ourselves directly with the servers on which these changes are physically implemented.

Each change is associated with timing constraints on when they can be executed, which are derived from allowed downtimes of the applications, as discussed next. We associate with an application $i \in A$ a set of allowed downtimes $T_i = \{(a_i^1, b_i^1), (a_i^2, b_i^2), \ldots\}$ (where $a_i^m < b_i^m < a_i^{m+1}$, for all m), which is usually a periodic pattern repeated on a daily or weekly basis (e.g., 2am to 4am every day). Note that scheduling a change of application i also affects applications in its conflict set A_i, which are using servers in S_i (that remain unavailable during this change implementation). Therefore scheduling of a change of application i must also take into account the allowed downtimes of the applications in A_i. The *permissible time set* for (changes of) application i, i.e., times at which a change of application i can be implemented, is thus represented as $\tilde{T}_i = T_i \cap_{i' \in A_i} T_{i'}$.

Figure 2 shows a system model that is detailed enough to capture all complexities in the scheduling question we have described so far. Here, applications are shown as a part of the conflict graph (i.e., applications are modeled as nodes in the graph, and conflicts between applications are represented by edges in the graph). Each node is associated with a queue[1] that buffers pending change re-

[1]We assume that changes of an application are served in a first-come-first-serve basis (although this assumption is not fundamental, and can be relaxed).

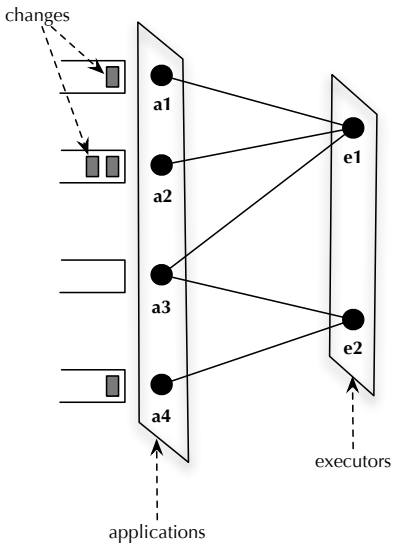

Figure 3. Bipartite graph showing the relationship between applications and executors, and the change queue of each application, for the system in Figure 2 and two executors (e_1 and e_2).

quests for that application. Associated with each application i is a permissible time set $\tilde{T}_i$, as described above; changes of application i must be scheduled at a time that falls within $\tilde{T}_i$. As an example, in Figure 2, let the allowed downtimes of applications a_1, a_2, a_3, a_4 respectively be 12am−3am, 1am−4am, 2am−5am, 3am−6am, every day. Then $\tilde{T}_{a_1} = \{1\text{am} - 3\text{am every day}\}, \tilde{T}_{a_2} = \{2\text{am} - 3\text{am every day}\}, \tilde{T}_{a_3} = \{2\text{am} - 4\text{am every day}\}, \tilde{T}_{a_4} = \{3\text{am} - 6\text{am every day}\}$.

Our change management system is also associated set of change executors E, where $E_i \subseteq E$ represents the set of executors that are capable of implementing changes of application $i \in A$. Each executor j is associated with an *availability time set* U_j of intervals when j is available, i.e., $U_j = \{(\bar{a}_j^1, \bar{b}_j^1), (\bar{a}_j^2, \bar{b}_j^2), \ldots\}$, where $\bar{a}_j^m < \bar{b}_j^m < \bar{a}_j^{m+1}$, for all m. In general, some executors may be capable of executing changes of multiple applications, while some changes can be implemented by multiple executors. In view of this, the change scheduling question also involves finding an *assignment* between applications and executors. Figure 3 shows the applications and executors being modeled as the two vertex sets of a bipartite graph, where an edge (i, j), $i \in A, j \in E$, represents the fact that changes of application i can be executed by executor j. In this model, therefore, the set of changes scheduled at any time must correspond to a matching in the bipartite graph. Note that a change of application i can be executed by executor j only at times that fall within $\tilde{T}_i \cap U_j$. In the example in Figures 2 and 3, if we assume that $\tilde{T}_{a_1} = \{1\text{am} - 3\text{am, every day}\}$ (as in the above example), and executor e_1 is available only Mon-Wed every week, then changes of application a_1 can only be executed (the executor can only be e_1 in this case) during 1am − 3am, Mon-Wed every week.

3.2. Objectives and Solution Methodology

Due to the typical critical nature of the changes, we would ideally like to implement the changes as soon as possible. In other words, we would like to schedule so as to minimize the average *change delay*, which represents the average delay in implementing changes in the system. The delay of a particular change implementation is measured as the time from the arrival of the change to the time the change is completed.

Minimizing average change delay is a very difficult problem, and its NP-hard even if all change arrival times are known in advance (offline problem). Even if statistics of the change arrival process is assumed to be known, the optimal dynamic change scheduling problem represents a very complex stochastic decision question, and the corresponding dynamic programming solution approach would be too complex to be practically useful. Since we are considering an online setting, it is worthwhile to consider another closely related, weaker (but nevertheless important) objective, namely that of attaining the *change capacity* of the system. The schedule that is optimal in those terms, however, corresponds to solving an NP-hard problem, even under additional assumptions on our system model.

Since finding the minimum delay and maximum capacity schedules is computationally difficult, we discuss and evaluate several greedy like heuristics for the problem. While some of these can be viewed as graph coloring based approximations (motivated by the close relationship of the offline change delay minimization problem to graph coloring), others can be viewed as approximations of the online scheduling solution that is capacity-optimal in certain special cases.

4. Solution Approach

In this section, we first argue that obtaining the optimal solutions to the minimum change delay and maximum change capacity questions require solving NP-hard problems, even under considerable simplifications of the problem constraints. We then describe some heuristics solution approaches that are motivated by these optimal solutions.

We first consider the offline change delay minimization problem. Consider a system in which there is a one-to-one mapping between the applications and executors, i.e.,

changes of any application $i \in A$ can be executed by exactly one executor, and each executor can execute changes of a single application only. Thus $|A| = |E|$, $|E_i| = 1 \forall i$, and $E_i \cap E_j = \phi$. Now assume that there is exactly one change per application, and all these changes arrive at time $t = 0$. Also assume that there are no timing constraints on the changes or the executors. In this scenario, the change delay minimization problem reduces to the minimum *sum-coloring* problem in the conflict graph G. Since the minimum sum-coloring problem is NP-hard [4], this implies that the offline change delay minimization problem is NP-hard as well.

The optimum online change delay minimization problem is in general a difficult question, and to the best of our knowledge, this question has not been addressed in the existing literature, either from competitive analysis (that makes no assumptions on the statistics of the arrival pattern) or from stochastic optimization (that assumes knowledge of arrival process statistics) perspectives.

The change capacity maximization problem is more tractable, and important special cases of this problem have been considered in the wireless networking literature[2] [5, 3]. Assume that time is slotted, and the execution time of each change equals one time slot. Moreover, assume that new schedules are computed after intervals of T slots, and the schedule computed at the beginning of an interval is used for the rest of the T-slot interval. In this set-up, assuming that there are no constraints on the permissible times of the applications or the availability times of the executors, the capacity-optimal change scheduling problem becomes similar to the transmission scheduling problem in a multi-channel access-point wireless network, considered in our earlier work [3]. The schedule, however, corresponds to choosing a maximum queue-length weighted independent set in the application conflict graph [3], such that it can be supported by the set of executors in a single time slot. Since the maximum weighted independent set problem is NP-hard, computing the capacity-optimal schedule as described above is also NP-hard in general.

In view of the difficulty of formulating and computing the optimal solutions, we present and study four heuristic solutions approaches for the change scheduling problem. These heuristic algorithms are run periodically, after regular intervals of time (T). In any run of an algorithm, we first remove from consideration all applications that do not have any changes waiting to be scheduled. We then iteratively pick an application (that does not conflict with an application whose change has already been scheduled), and schedule it's head-of-line change (i.e., earliest-to-arrive change, since the the change queues are FIFO) with an executor that can execute it in the minimum possible time. Also note that a change is only scheduled if it can be started before the next scheduling instant, i,e., within a time T from the current time. The criteria for choosing the heuristic depends on the exact algorithm considered, as discussed next. Note the simple greedy nature of the algorithms; while this results in low implementation complexity, we show in the next section that they perform well, and quite close to the optimum in some special cases.

In the first heuristic, called **random scheduling**, we pick each application randomly, from all "schedulable" applications, i.e., non-conflicting applications that have not been scheduled already. In the second heuristic, called **degree scheduling**, we pick the schedulable application that has the largest degree in the *residual* conflict graph. The residual conflict graph at any iteration is constructed by removing all nodes (and the edges incident on them) that correspond to applications that are non-backlogged, have already been include in the schedule, or have been removed from consideration due to conflict with a scheduled node (application). These two algorithms are motivated by randomized and degree vertex coloring algorithms. Note that the degree coloring algorithm in a popular graph coloring heuristic that is guaranteed to color a graph with degree Δ in at most $\Delta + 1$ colors [2].

In the third heuristic, which we refer to as **longest queue scheduling**, at any iteration we pick the schedulable application with the longest change queue. This heuristic is a greedy version of the change scheduling policy based on maximum queue-length weighted independent set, as discussed earlier, that is capacity-optimal in certain special cases. In the fourth heuristic, called **earliest time scheduling**, at any iteration we pick the schedulable application whose head-of-line change has the earliest arrival time. Below we describe the longest queue scheduling algorithm; the application selection criterion in step 2(i) of the algorithm can be easily modified to suit the description of the other three heuristic algorithms, by incorporating the appropriate metric instead of longest queue.

Longest Queue Scheduling

At every time slot, do the following:

1. Remove from consideration all applications for which the permissible time set does not include the current time slot, and all executors whose availability time set does not include the current time slot.
2. Iteratively do the following until all applications or executors are exhausted:
   (i) Pick the application with the longest queue and schedule it with a randomly chosen executor that can execute the application.
   (ii) Remove from consideration the chosen application and the corresponding executor, as well as all other applications that conflict with the chosen application.

[2]In this case, each application corresponds to a wireless transceiver sharing a single channel, and the application conflict graph corresponds to the wireless interference graph.

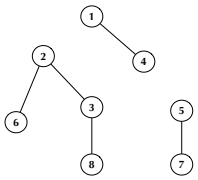

Figure 4. Conflict graph: Edges represent scheduling conflicts between the applications (1, 2, ..., 8).

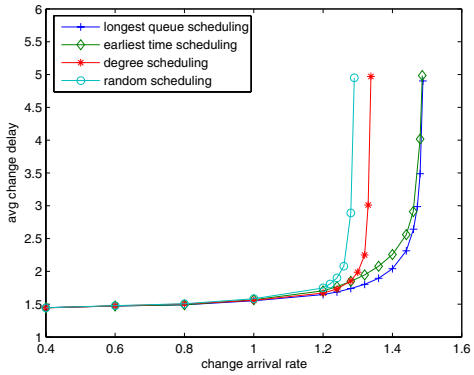

Figure 5. Change delay comparison.

5. Simulation Evaluation

In this section, we evaluate the performance of the algorithms described in the previous sections, in terms of metrics like the change delay and change capacity. Since the delay or capacity optimal algorithms are practically infeasible, we only simulate and study the performance of the random, degree, longest queue and earliest time algorithms.

5.1. Simulation setup

In the representative simulation results shown below, simulations are carried out in a system with 8 applications and 4 executors. The conflict graph that we use (except in the simulations where we vary the conflict degree) is shown in Figure 4. We assume in general that each application is associated with two executors, and an executor is capable of handling the changes of only one application; this assumption is relaxed when we study the effect of degree of executor cross-training, where we allow an executor to handle changes of multiple applications. In our simulation setup, changes arrive at the change management system in a continuous manner, according to a random process. Each change is assumed to have an execution time of 6 hours.

Table 1. Change capacity.

random	degree	longest queue	earliest time
1.292	1.340	1.490	1.488

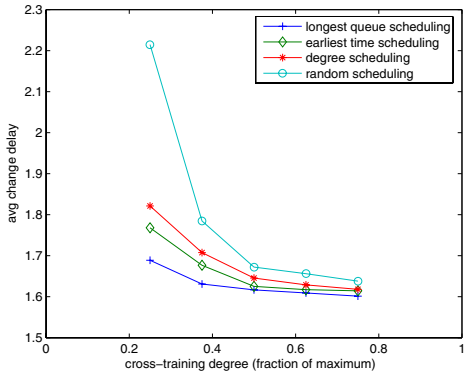

Figure 6. Effect of executor cross-training.

Scheduling of changes is done on a daily basis, and changes that are not scheduled in the same day, and queued up, and considered for scheduling in subsequent days. The timing constraints on the applications and the executors are chosen at random. All delay results shown are obtained by averaging over at least 100,000 days, to ensure that the values obtained correspond to their steady state values.

5.2. Change delay and capacity

Figure 5 show the average change delay in the system, as the rate of change arrivals is increased. Table 1 shows the change capacity attained in the system (i.e., the value at which the delay "blows up" in Figure 5). Figure 5 shows that the change delay increases smoothly as the arrival rate increases, as expected, till the change capacity of the system is reached. We observe that while all the algorithms perform similarly at low values of the change arrival rate, the average change delay curves for the different algorithms diverge as the arrival rate increases. The longest queue algorithms performs the best, closely followed by the earliest time algorithm. The degree and random scheduling algorithms perform much worse, random scheduling being the worst. Table 1 shows that the capacities obtained by the various algorithms follow a similar trend.

5.3. Effect of executor cross-training

Figure 6 shows the variation of average change delay, as the degree of cross-training of executors increases, while all other parameters remain fixed. The cross-training degree

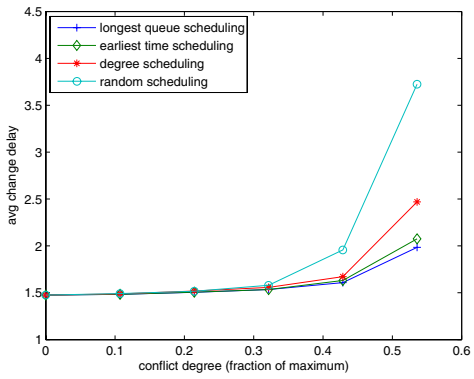

Figure 7. Effect of degree of conflicts.

is computed as the average number of applications that an executor is capable of handling. The x-axis shows the cross-training degree expressed as a fraction of the maximum value (which is 8, since there are 8 applications in this case). The comparative performance of the algorithms is similar to that observed in Figure 5. We observe that the average delays improve as the cross-training degree increases, as expected; however, the marginal benefits of cross-training show "diminishing returns" after the cross-training degree exceeds about 50% of the maximum, and the curve is nearly flat at these values.

5.4. Effect of degree of conflicts

Finally, we study the effect of the degree of conflicts in the system on the overall delay attained. For a system of 8 applications and 4 executors, where each application is associated with two executors (no cross-training), the change delay attained by different algorithms is shown in Figure 7 as a function of the degree of conflict. The degree of conflict is varied by changing the number of edges in conflict graph of the applications, where the conflict edges are chosen randomly. For example, in Figure 7, the conflict degree value (calculated as a fraction of the maximum) of x corresponds to the case where $x \times 28$ edges are added at random in the conflict graph; note that $\binom{8}{2} = 28$ is the total number of possible edges in the graph. The results shown is obtained by averaging over 10 runs for the same value of the conflict degree. The results show that the average change delay does not increase considerably till the conflict degree reaches about 0.4 (for the random scheduling algorithm, change delay starts deteriorating at a slightly smaller value of the conflict degree), after which it tends to increase sharply with further increase in the conflict degree. The comparative performance of the algorithms is similar to that observed in the earlier simulation results; the longest queue algorithm performs the best, as in the previous cases.

6. Discussion and Conclusion

In this paper, we consider the issue of effective change management for Enterprise IT services. We model the change approval step in the change management process and address the question of optimal scheduling of changes, with the goal of minimizing the average change delay and maximizing the overall change capacity of the system. Our simulation based performance study yields several critical observations. We observe that a longest-queue based greedy heuristic policy performs the best among all four heuristic solution approaches considered. The performance of a policy that picks the change to be scheduled randomly performs the worst; its performance is significantly worse than that of the longest-queue based policy both in terms of average change delay (particularly at high change arrival rates) and change capacity. Our results indicate that the effect of average degree of scheduling conflicts between applications is negligible till about 40% of the maximum, beyond which the average change delay increases sharply with an increase in the average conflict degree. This suggests that the conflict degree between applications must be kept below this critical level during the design of Enterprise IT services management system, even at the cost of increasing the number of servers that implement these applications. Our studies also indicate that the average change delay improves significantly as the degree of cross-training of executing personnel increases to about 50% of the maximum value, beyond which point increased cross-training does not have a significant effect on process performance. These observations lead us to suggest re-engineering of the change management process so that the longest-queue based algorithm for scheduling changes, and the degree of application conflicts and cross-training of executing personnel is maintained at the level of 40-50%.

References

[1] Directory of software & services for ITIL and ITSM. *http://www.itil-itsm-world.com/*.

[2] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill Book Company, 1990.

[3] K. Kar, X. Luo, and S. Sarkar. Throughput-optimal scheduling in multichannel access point networks under infrequent channel measurements. *Proceedings of IEEE Infocom 2007*, Anchorage, AK, May 2007.

[4] E. Kubika and A. Schwenk. An introduction to chromatic sums. *Proceedings of the 17th Annual ACM Comp. Sci. Conf.*, Louisville, KY, Feb 1989.

[5] L. Tassiulas and A. Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE Transactions on Automatic Control*, 37(12), Dec 1992.