ReLauncher: Crowdsourcing Micro-Tasks Runtime Controller

Pavel Kucherbaev, Florian Daniel, Stefano Tranquillini, Maurizio Marchese
University of Trento - DISI
Via Sommarive 9, 38123 Povo (TN), Italy
{lastname}@disi.unitn.it

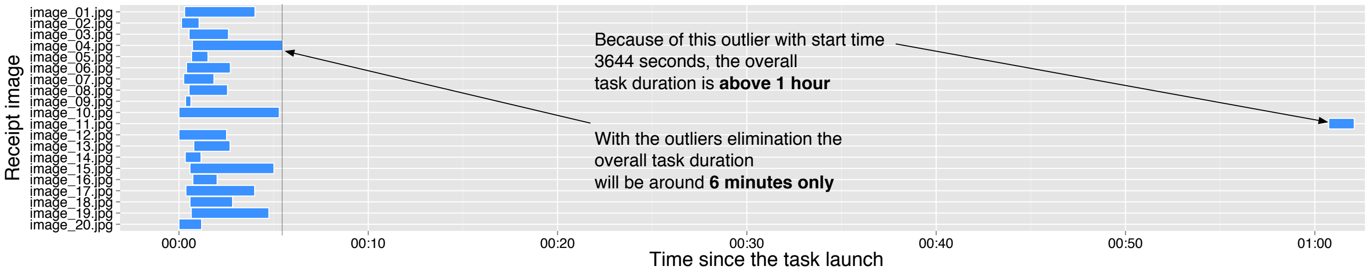


Figure 1. The timeline of the receipt transcription task launched on CrowdFlower with 20 data units (receipt photos). Each data unit is assigned to a single worker. The assignment on the very right for the unit "image_11.jpg" was finished only in 1 hour after the launch of the task, because the crowdsourcing platform kept it reserved for a certain time for a worker who left it without finishing it.

ABSTRACT

Task execution timeliness, i.e., the completion of a task within a given time frame, is a known open issue in crowdsourcing. While running tasks on crowdsourcing platforms a requester experiences long tails in execution caused by abandoned assignments (those left by workers unfinished), which become available for other workers only after some expiration time (e.g., 30 minutes in CrowdFlower). These abandoned assignments result in significant delays and a poor predictability of the overall task execution time. In this paper, we propose an approach and an implementation called ReLauncher to identify such abandoned assignments and relaunch them for other workers. We evaluate our implementation with an experiment on CrowdFlower that provides substantive evidence for a significant execution speed improvement with an average extra cost of about 10%.

Author Keywords

Crowdsourcing; Micro-tasks; Execution speed; Task relaunching

ACM Classification Keywords

H.5.3. Group and Organization Interfaces: Computer-supported cooperative work

CSCW '16, February 27–March 02, 2016, San Francisco, CA, USA
Copyright © 2016 ACM. ISBN 978-1-4503-3592-8/16/02...$15.00.
DOI: http://dx.doi.org/10.1145/2818048.2820005

INTRODUCTION

Crowdsourcing is the outsourcing of a unit of work to a crowd of people via an open call for contributions [8]. Thanks to the availability of online crowdsourcing platforms, such as Amazon Mechanical Turk (MTurk), CrowdFlower and many others [12, 14], the practice has experienced a tremendous growth over the last few years [11]. Crowdsourcing demonstrated its viability in a variety of different fields, such as data collection and analysis or human computation – all practices that use so-called *micro-tasks*, which ask workers to complete simple assignments (e.g., label an image or translate a sentence) in exchange for an optional reward (e.g., few cents or dollars).

A *requester* can crowdsource work by publishing a *task* (a so-called "HIT" in MTurk or "job" in CrowdFlower). The requester can upload a *dataset* for the task and ask one or more (W) crowd *workers* to process each *data unit* from the dataset. For each data unit W corresponding *assignments* are created that bind together the task and the data units, as well as the workers that accept the assignment and their eventual results. Each assignment can be in one of 3 possible states (see the thick, solid sub-model in Figure 2 that provides a Petri Net model expressing the execution semantics of assignments): *to be assigned* – no worker joined the assignment yet, *started* – a worker is working on the assignment, *finished* – the worker submitted results for the assignment. The assignment *duration* is the time from when the assignment is started to when it is finished. If an assignment is started but not finished within a given timeout, the assignment *expires* and becomes again available to other workers. The worker of the expired assignment is no longer able to submit results and, hence, is not rewarded.

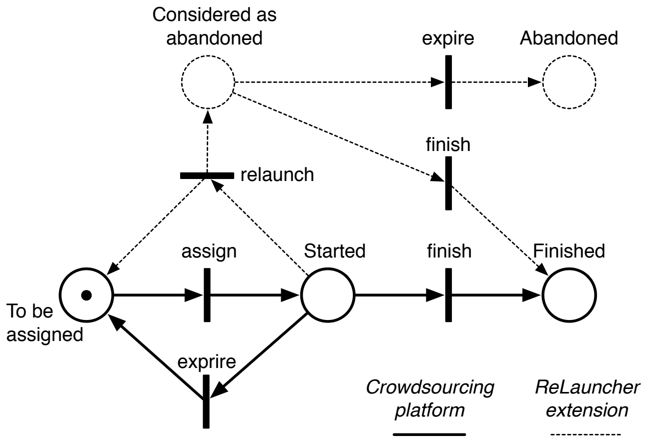


Figure 2. Petri Net modeling the execution semantics of assignments when executed with ReLauncher (dashed) integrated into a crowdsourcing platform (solid). Tokens correspond to assignments to be processed by workers.

In our past experiments on CrowdFlower, we identified that indeed sometimes workers leave assignments unfinished. While in MTurk a worker can preview an assignment before officially accepting it and start working on it, in CrowdFlower selecting an assignment (based on title and short description only) is already interpreted as the worker accepting the assignment. At this point, the worker has four options: 1) to work on it, 2) to push the button "Give up" to state that he/she does not want to work on it and to make it again available to other workers, 3) to navigate away from the browser tab, or 4) to simply close the browser tab. There are various reasons why workers may not want to work on a given assignment, e.g., they are not sure they understood the assignment correctly (e.g., due to a sloppy description of the assignment), they consider it too complex, they believe they would not perform well (e.g., because a blurred image is given for text transcription), they are distracted by external events (e.g., a phone call or children), they simply got bored and need a break, and similar. The title of the button "Give up" is misleading and discouraging, so workers usually just close the browser tab, leaving the assignment pending till it expires on its own.

We call this kind of started but not finished assignments *abandoned*. Abandoned assignments stay in the *started* state for a defined period of time (fixed as 30 minutes in CrowdFlower and as 1 hour but adjustable in MTurk), during which no other workers can start working on the same assignment. This pattern is for instance visible in Figure 1, where the assignment for the image "image_11.jpg" (on the very right) expired twice, which resulted in the assignment being finished only after 1 hour. This is the execution pattern that leads to the problem of long tails in the overall execution of a task.

In this paper, we propose an approach to identify likely abandoned assignments during runtime so as to make the corresponding data units available to other workers earlier and, eventually, to speed up the overall execution time of tasks. We describe ReLauncher, an implementation of the approach for CrowdFlower, and report on it's performance.

RELAUNCHER

Without knowledge of the behavior of workers, it is not possible to unequivocally identify abandoned units, as the abandoning takes place in the worker's browser, i.e., the client side, an environment we don't have access to. We thus simplify the challenge of identifying abandoned assignments as the challenge of identifying a maximum assignment duration *MaxDur* beyond which we consider assignments as abandoned. Differently from the common static timeout of assignments, we calculate *MaxDur* dynamically during runtime taking into account the real dynamics of a task.

Approach

The dashed model part in Figure 2 illustrates the new execution semantics of assignments. ReLauncher periodically checks if started assignments were started earlier than *MaxDur* time ago. Identified slow assignments are *relaunched*, which means that they are moved to a *considered as abandoned* state, while a copy of the assignments is created in the *to be assigned* state, making them again available to other workers. If the assignments considered as abandoned are really abandoned by the worker, they expire and move to the *abandoned* state without making them again available to other workers. While if these assignments are instead just delayed, the workers eventually finish them, their contribution is registered, and they are rewarded. With the duplication of the assignment ReLauncher tries to speed up the execution of the assignment, as there are now two workers that work in parallel on the same assignment.

From our past experiments conducting tasks (receipt transcription with 300 data units) on CrowdFlower we identified that assignment durations (Figure 3A) follow a log-normal distribution (p-value = 0.275 and p-value = 0.818 if only durations between 1% and 99% percentiles are taken). Because the assignment start time in average is short (workers start tasks after they are published in a matter of seconds) compared to the assignment duration time, there is a correlation (0.627) between the order in which the assignments are finished (assignment index) and the assignment duration time. The black dots on Figure 3B represent local maximum durations before a given assignment index. In this paper, our goal is not to model assignment durations precisely, but to find a simple way to eliminate assignments with very long durations. We thus compute a linear regression from the local maximums and obtain a slope = 1.667 and an intercept = 32.839, with a p-value close to 0 (the black straight line in Figure 3B). This means assignment durations grow monotonically with their index, and we can estimate the slowest allowed duration *MaxDur* as the value of the linear regression maximum index (in our case index = 300), which in our experiment reported in figure 2B predicts a *MaxDur* of 533 seconds. Algorithm 1 formalizes the technique.

The slope of the cumulative distribution of the assignment completion times (Figure 3C) indicates the speed of the task completion. There is a moment where the speed starts slowing down (tangent B). This would be the right time to identify abandoned assignments and start relaunching them. Before that time, it is not necessary to intervene, as assignments are

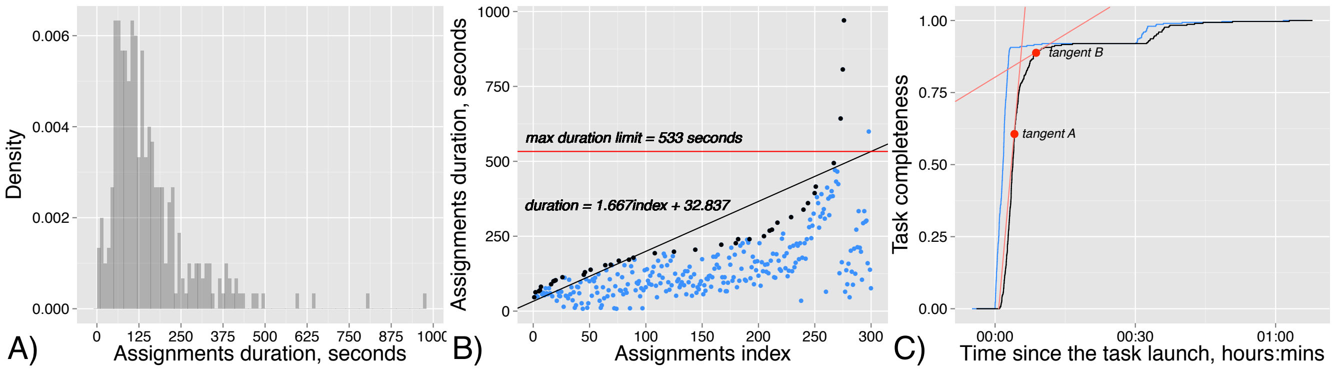


Figure 3. The runtime behavior of crowdsourced receipt transcription task with 300 data units to be processed: A) assignment duration distribution (log-normal), B) assignment duration dependency on index of assignment, C) cumulative assignments start (blue) and completion (black).

Algorithm 1 Last Assignment Duration Estimation

```
procedure ESTIMATEMAXDURATION($durs, N, W$) ▷ $durs$ – an array of durations of already finished assignments, N – number of data units, W – number of workers requested to process each data unit
    $maxValues \leftarrow array()$
    $maxIndexes \leftarrow array()$
    $i \leftarrow 1$
    while $i < length(durs)$ do
        if $(length(maxValues) == 0$ or $durs[i] > maxValues[length(maxValues)])$ then
            $maxValues.push(durs[i])$
            $maxIndexes.push(i)$
        $i \leftarrow i + 1$
    $lr \leftarrow linearRegression(maxValues, maxIndexes)$
    $slope \leftarrow lr.slope$
    $intercept \leftarrow lr.intercept$
    $MaxDur \leftarrow N \times W \times slope + intercept$
    return $MaxDur$
```

already completed speedily. Also, if we start too early we may classify more assignments as abandoned than necessary (extra cost); if we start too late we may delay the overall task execution too much. From our experiments on CrowdFlower we know that execution usually starts slowing down around 85% of task completion, while this percentage was never below 75%. In order to be on a safe side, in the following we use a heuristic approach and start relaunching assignments after 70% of task completion.

ReLauncher could relaunch a given assignment infinite times, since a relaunched assignment can be relaunched again, again, and again (Figure 2). In order to prevent an infinite relaunching and significant extra costs, we stop relaunching assignments for a same data unit (i) when at least W assignments are finished (the unit was processed as expected), or (ii) when a given relaunching budget is reached (despite relaunching, the unit could not be processed).

Implementation

We implemented ReLauncher as described above for CrowdFlower as a web application using AngularJS for the front-end, ExpressJS for the back-end and R for data processing and graph generation. The application is open-source and publicly available[1] for the benefit of the community.

On CrowdFlower a requester cannot obtain information about individual started assignments but only observe the state of data units. A data unit can be in one of 4 possible states: *available* – it can still be assigned to workers, *assigned* – all W assignments were created and new workers can no longer join, *processed* – all assignments are finished and *cancelled* – the data unit can not be assigned to any workers and it is no longer expected to be processed. While it is technically possible to make a data unit available again for new assignments by rejecting finished ones, this would be unfair and unethical, as there is no evidence a given worker violated any rule. We therefore simply relaunch data units, rather than assignments. That is, when we see that a unit is in the *assigned* state for a time longer than *MaxDur* we inactivate the unit (we "cancel" it, in CrowdFlower terminology), which prevents the platform from expiring respective assignments and making them available again while still allowing workers to finish their ongoing assignments, and append a copy of the data unit to the dataset other workers can work on.

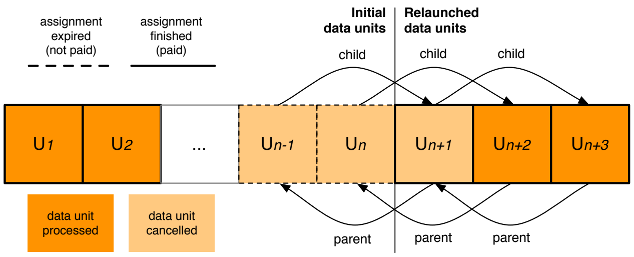
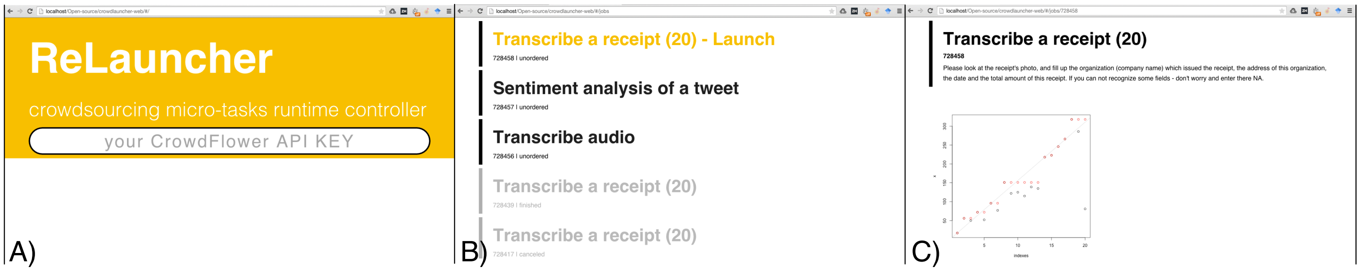


Figure 5. The chains of data units created by ReLauncher.

In order to keep track of the relaunching dynamics, we maintain relaunched data unit chains (Figure 5), where a newly created unit has a "parent" link to the original unit, and the original unit has a "child" link to the new unit. As soon as the W_{th} assignment of a given unit is finished in a chain, we use the "parent" and "child" links to cancel all related units.

[1] https://github.com/pavelk2/crowd-relauncher


localhost/Open-source/crowdlauncher-web/#/
ReLauncher
crowdsourcing micro-tasks runtime controller
your CrowdFlower API KEY
A)
localhost/Open-source/crowdlauncher-web/#/jobs
B)
localhost/Open-source/crowdlauncher-web/#/jobs/728458
C)
x
50
100
150
200
250
300
5
10
15
20
indexes

Transcribe a receipt (20) - Launch
728458 | unordered

Sentiment analysis of a tweet
728457 | unordered

Transcribe audio
728456 | unordered

Transcribe a receipt (20)
728439 | finished

Transcribe a receipt (20)
728417 | canceled

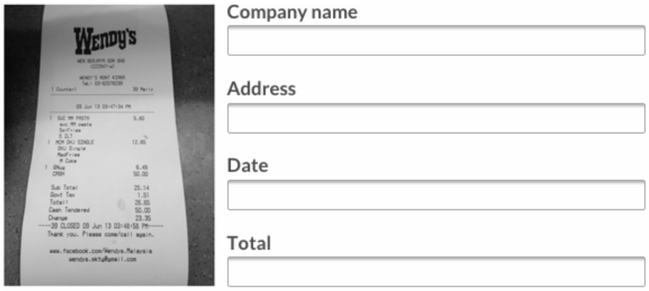
Transcribe a receipt (20)
728458
Please look at the receipt's photo, and fill up the organization (company name) which issued the receipt, the address of this organization, the date and the total amount of this receipt. If you can not recognize some fields - don't worry and enter there NA.

Figure 4. Screen shots of the ReLauncher implementation: A) authorization page, B) task listing page, and C) launched task information page.

In order to use ReLauncher, requesters need to provide ReLauncher with their CrowdFlower API key (Figure 4A). This key is not stored anywhere in the system and has to be entered every time a requester uses ReLauncher. When a key is inserted, ReLaucher displays the list of tasks (Figure 4B) created by the requester on CrowdFlower. The requester can launch any task that is not yet completed. When a task is launched, ReLauncher shows the task information (Figure 4C) along with a graph plotting the current finished assignments and an estimated maximum assignment duration time.

EVALUATION

We deployed the same task with and without using ReLauncher, repeating the experiment 5 times for better robustness of the study. The task asked workers to transcribe a receipt by filling four textual fields (company name, address, date of purchase, total amount) given photo of the receipt (Figure 7) for a reward of USD 0.10. Each task contained 100 receipt photos (100 data units), each receipt photo needed to be transcribed by a single worker ($W = 1$). The total cost of the experiment was USD 138.48.



Company name

Address

Date

Total

Figure 7. The user interface of the receipt transcription task.

The cumulative executions of the experiment are shown in Figure 6, where the blue lines represent started assignments and the black lines represent finished assignments. In the same figure: $t_{completion}$ – the overall task completion time in seconds, $A_{relaunched}$ – the number of relaunched assignments, and $A_{extrapaid}$ – the percentage of extra assignments paid because of relaunching. The aggregated results are given in Table 1, where μ is the mean value and s the standard deviation. Workers on CrowdFlower can leave a feedback to a task. Our task received consistently good marks above 4 out of 5.

The collected data show that with ReLauncher tasks are completed more than 3 times faster (large effect size according to Cohen's d statistic: $d = 2.91$), with an average extra cost of around 10.4%. A Welch's t-test shows that the mean task completion time with ReLauncher is shorter than without ($p = 0.004$, $df = 4.432$). In order to obtain this result, an average of 15.8% of the assignments were relaunched, many of them still producing results in the end (the 10.4% that determine the extra cost). This leads to two immediate conclusions: while the approach described in this paper provides substantive benefits in terms of time reduction and there is still room for improvement, decreasing the amount of false positive relaunches may further decrease the extra cost.

Earlier we explained when we start relaunching assignments (after 70% of assignments are completed). For the sake of this paper, we applied this simple heuristic based on historical data. It is however clear that identifying with higher precision the best moment when to intervene may further save both time and money. It is however good to note that already with 70% of assignments completed, we are able to compute good estimates of the maximum task durations. Better identifying the time of intervention will provide the prediction algorithm with even more data.

As for the generalizability of the described approach, it is important to note that the whole work we present in this paper does not make any assumption regarding the type of task that is crowdsourced. All the ReLauncher needs is monitoring the runtime behavior of assignments (start and completion times, durations). The heuristic to start relaunching assignments after 70% of them have finished can be substituted with a function that is able to dynamically identify the turn point in Figure 2C. Further, the approach is based on pure statistics, and the linear regression model considers local maximum duration times. This means that also tasks with highly variable

	Without ReLauncher	With ReLauncher
Task completion time	μ = 3146 seconds s = 1037 seconds	μ = 948 seconds s = 241 seconds
Assignments relaunched	N/A	μ = 15.8 s = 6.9
Extra costs	N/A	μ = 10.4% s = 5.07%

Table 1. Performance comparison of evaluation experiments without and with ReLauncher.

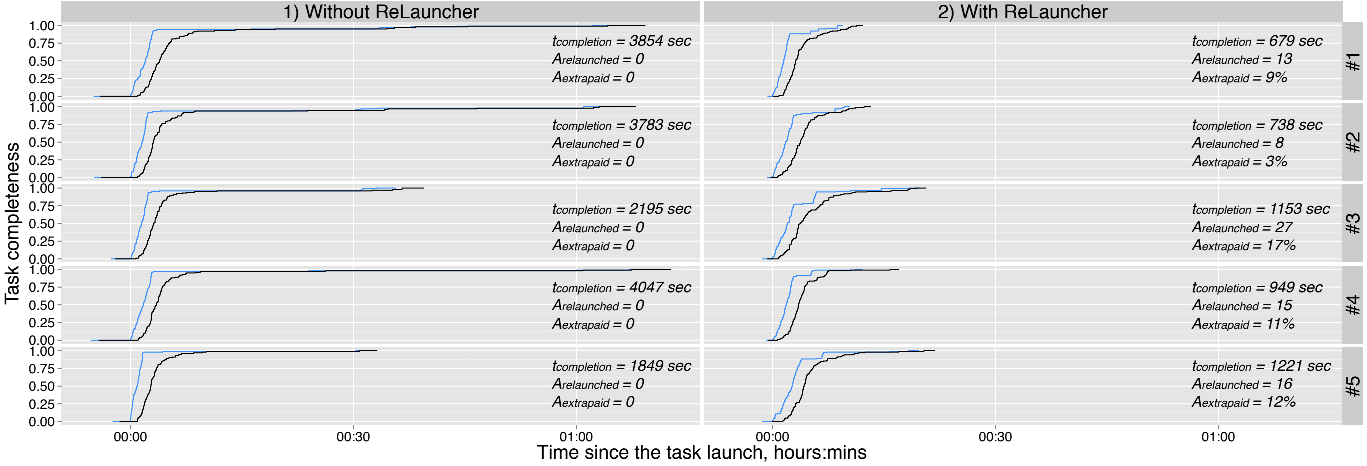


Figure 6. Cumulative assignments start (blue) and assignments completion (black) without and with ReLauncher.

assignment durations (e.g., tasks with variable lengths of data units) can be managed. Of course, the higher the variability, the higher the estimated $MaxDur$.

Also, in principle, it could be an option to maintain a knowledge base of regression models for different task types, e.g., not to have to compute the model at runtime, but this would be helpful only if we had to start relaunching tasks already in the very beginning of the overall task execution where there are not yet many assignments finished to compute the model from. But we have shown that it does not make sense to intervene before 70% or more assignments have completed, which in turn provides ReLauncher with enough data and time to compute task-specific regression models on the fly.

As a last consideration, we would like to emphasize that ReLauncher does not penalize any worker and that by no means it prevents workers that seriously work on an assignment from submitting results and obtaining the respective reward. We admit that for instance tasks that require significant network activity (e.g., collecting URLs or content from many different web pages) could potentially result in considering assignments by workers with slow network connections as abandoned, while instead they are still working. However, as said, this would happen without any side-effect on the workers and only cause the ReLauncher to manage more relaunched assignments. However, it is also intuitive that workers with slow network connections would very likely not choose to work on tasks that ask for significant networking activity.

RELATED WORK

Researchers tried various approaches to improve task execution speed, such as different motivation strategies for workers or introducing interventions to the execution process itself, also aiming at crowdsourcing complex work.

In [7] the authors identified that on MTurk an increased reward actually decreases the demand for the task, as workers perceive the task as more complex. The authors identified that the task completion time monotonically decreases for increasing reward values. Also, those working on tasks with higher rewards perceive the value of their work to be greater, which does not motivate them more than workers working on tasks with lower rewards [10]. On MTurk, the tasks with a high amount of units attract most workers [5, 6]. Still there is a problem of long tails in task execution, as also these tasks attract less workers towards their end. The problem can be solved, for example, by paying an extra bonus to workers who finish a predefined amount of assignments [6]. Another approach to address the long tails is by adjusting the tasks reward according to the amount of not finished assignments a task has left and the time passed since its launch [4]. A survival analysis based model is proposed to predict tasks completion times on MTurk based on various task and marketplace attributes [13]. This model is used in [7] to predict what reward amount to set for a task to have it completed by a desired time. In [1] the authors proposed a retainer model paying workers a small reward to keep them waiting and to respond to a real task as soon as it becomes available. With ReLauncher we do not provide any extra motivation for workers to speed up execution of individual assignments; instead, we intervene at runtime to relaunch assignments that we identify as abandoned, which would slow down the overall task execution time.

Works that specifically focus on execution time are the following: TurkPrime (`https://www.turkprime.com/`) has a task restart feature that puts tasks back on top of the tasks listing page on MTurk, still it requires direct involvement of the requester to monitor and manually restart tasks. In [3], the authors allow the requester to define a set of rules that, depending on the workers' performance, can trigger different actions, adjusting the execution process according to the requester's requirements. A sensible design of low-level rules could be able to achieve a functionality similar to ReLauncher. Then there are approaches, such as TurKit [9], that allow a requester to program a logic/algorithm to execute multiple tasks. While these approaches help crowdsourcing work easier, they do not address the problem of delays in tasks execution. REACT [2] is a system that dynamically assigns tasks to different workers in order to meet timeliness requirements and to make sure that given quality standards are met. Yet, the approach proposes the matching of workers with tasks based on worker profiles, which only the operators of crowdsourcing platforms have access to and requesters cannot benefit from. ReLauncher, instead, runs with commonly available information only and, more importantly, does not require any additional input from the requester.

LIMITATIONS AND FUTURE WORK

ReLauncher is a simple yet tangible contribution to the state of the art in crowdsourcing: it is able to effectively reduce execution times of tasks without any additional configuration or input from the requester without affecting the work as perceived by workers. The effect of ReLauncher was statistically significant in the described experiment conducted with CrowdFlower. ReLauncher showed a 3-times improvement of the overall task execution time at an average extra cost of 10%. The approach works as a client of state-of-the-art crowdsourcing platforms (using their APIs), but it could easily also be incorporated directly inside the platforms.

Limitations

The current implementation is limited to CrowdFlower, and the evaluation is based on one task type only (transcribing receipts). In order to ascertain the generalizability of the approach, additional experiments with different task types would be helpful. While we did some additional experiments with the same task using fewer than 100 data units (obtaining similar results as described in this paper), we did not test how ReLauncher works with bigger datasets. Also, we neither used test questions to select workers nor did we assess the quality of results in our experiments, as assessing quality is a research topic in its own that we would like to investigate further in our future work. All experiments were conducted with a fixed reward of USD 0.10; we did not further study the effect of varying reward amounts. Finally, it is important to acknowledge that ReLauncher is not able to compensate for the lack of task selection due to poor task design.

Future work

Next, we plan to conduct other experiments to test ReLauncher with different task types, different dataset sizes, and different algorithms for the identification of when to start relaunching assignments. We also plan to try to collect client-side worker activity data, e.g., by injecting JavaScript code into the task user interface, to enable ReLauncher to know exactly whether a worker is active or not and to increase the efficiency of the relaunching algorithm. If we were able to relaunch only those assignments that were really abandoned, ReLauncher could operate at zero extra cost. We also would like to study whether it is possible to use the runtime controller to identify workers who do not perform tasks as requested. For example, we will specifically look into the quality of those assignments that are on the left side of the durations distribution (are they fast because workers did not follow all instructions?). Of course, it would also be useful to get in touch with workers to investigate also qualitatively the reasons for their performance, spanning from very slow to very fast executions and from low to high quality. Similarly, it would be interesting to study which requesters running which types of tasks could benefit most from ReLauncher.

ACKNOWLEDGEMENTS

The authors thank Wil Stevens from CrowdFlower and the hundreds of workers who participated in the experiments.

REFERENCES

1. Michael S. Bernstein, Joel Brandt, Robert C. Miller, and David R. Karger. 2011. Crowds in Two Seconds: Enabling Realtime Crowd-powered Interfaces. In *UIST 2011*. 33–42.
2. Ioannis Boutsis and Vana Kalogeraki. 2013. Crowdsourcing under Real-Time Constraints. In *IPDPS 2013*. 753–764.
3. Alessandro Bozzon, Marco Brambilla, Stefano Ceri, and Andrea Mauri. 2013. Reactive crowdsourcing. In *WWW 2013*. 153–164.
4. Dana Chandler and John Joseph Horton. 2011. Labor Allocation in Paid Crowdsourcing: Experimental Evidence on Positioning, Nudges and Prices. In *HCOMP 2011*.
5. Lydia B. Chilton, John J. Horton, Robert C. Miller, and Shiri Azenkot. 2010. Task Search in a Human Computation Market. In *HCOMP 2010*. 1–9.
6. Djellel E. Difallah, Michele Catasta, Gianluca Demartini, and Philippe Cudré-Mauroux. 2014. Scaling-up the Crowd: Micro-Task Pricing Schemes for Worker Retention and Latency Improvement. In *HCOMP 2014*.
7. Siamak Faridani, Bjoern Hartmann, and Panagiotis G. Ipeirotis. 2011. What's the Right Price? Pricing Tasks for Finishing on Time.. In *HCOMP 2011*.
8. Jeff Howe. 2008. *Crowdsourcing: why the power of the crowd is driving the future of business* (1st ed.). Crown Publishing Group, New York, NY, USA.
9. Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. 2010. TurKit: Human Computation Algorithms on Mechanical Turk. In *UIST 2010*. 57–66.
10. Winter Mason and Duncan J. Watts. 2009. Financial Incentives and the "Performance of Crowds". In *HCOMP 2009*. 77–85.
11. Massolution. 2013. The crowd in the cloud: exploring the future of outsourcing. White Paper. (January 2013).
12. Donna Vakharia and Matthew Lease. 2015. Beyond Mechanical Turk: An Analysis of Paid Crowd Work Platforms. In *Proceedings of the iConference*.
13. Jing Wang, Siamak Faridani, and Panagiotis G. Ipeirotis. 2011. Estimating Completion Time for Crowdsourced Tasks Using Survival Analysis Models. In *CSDM 2011*. 31–34.
14. Man-Ching Yuen, I. King, and Kwong-Sak Leung. 2011. A Survey of Crowdsourcing Systems. In *PASSAT-SocialCom 2011*. 766–773.