Crowdsourcing in the Software Development Industry

Daniel Fried

friedaniel@gmail.com

Copyright 2010, Daniel Fried. All Rights Reserved. For copyright permissions, contact the author directly or Professor Amar Gupta at The University of Arizona (gupta@email.arizona.edu)
Abstract
The term crowdsourcing was coined by journalist Jeff Howe, who defines it as “outsourcing a task to a large group of people in the form of an open call”. Crowdsourcing has been used increasingly by the software industry to both lower opportunity costs and increase quality of output by utilizing capital from outside the company, in the form of the experience, labor, or creativity of outside programmers worldwide. Some platforms for crowdsourcing applications, such as Amazon’s Mechanical Turk, provide a means for the acquisition of large amounts of human knowledge in an inexpensive manner. Other platforms, such as TopCoder, use crowdsourcing methods to drive software coding and development, creating contests where programmers compete for a monetary prize by designing algorithms that meet the company’s specifications. A third type of platform, exemplified by MathWorks’s programming competitions, utilizes a unique form of “competitive collaboration” to produce highly efficient software with almost no financial cost to the project coordinators. This paper will investigate competitive and collaborative software frameworks for online crowdsourcing. Other topics investigated will be participants in the collaboration process, external and intrinsic incentives to ensure crowd participation, and parallel versus iterative design and development in crowdsourced applications.

Introduction – Open Source Software

The growth of the free and open source software (FOSS) movement in the 1980s built a foundation for the distributed development of software and the incorporation of design contributions from a diverse and geographically non-localized community of programmers (von Krogh and von Hippel 2003). The open-source community used the growing capabilities of the Internet to share software and code, coordinating the development of sophisticated open source projects such as the Apache Web Server and the Linux operating system through “user
innovation networks,” giving anyone with Web access the power to “download, use, modify, and further develop” the community’s software (von Hippel 2008). Open source’s economic model was a hybrid of private investment and collective action – programmers “used their own resources to privately invest in creating novel software code… then freely revealed it as a public good” (von Krogh and von Hippel 2003). By releasing the source code for their programs, the development communities lost their competitive edge with vendors of proprietary code such as Microsoft, but gained widespread adoption of their code and appreciation of its robust and easily modifiable qualities. FOSS demonstrated the capability of a distributed group to develop successful software, even when most of the contributors did not receive financial compensation for their labor. In these ways, the open source movement laid the foundation for the software development crowdsourcing platforms that will be discussed in this paper.

Micro-Crowdsourcing and Amazon's Mechanical Turk

Micro-crowdsourcing refers to the distribution of small tasks requiring little skill and time to complete. Monetary compensation for micro-crowdsourcing, if any is given, is typically small, in accordance with the undemanding nature of the work. While larger scale crowdsourcing projects may involve design or a creative process, micro-crowdsourcing views workers as human processors to complete massive amounts of simple jobs in parallel. This analysis will focus on Amazon's Mechanical Turk micro-crowdsourcing platform, one of the largest publicly available platforms with more than 100,000 workers. Amazon CEO Jeff Bezos describes Mechanical Turk as “artificial artificial intelligence” (Pontin), referring to its use as a tool in the implementation of artificial intelligence applications such as audio transcription, image tagging, and object tracking in computer vision systems (Corney). Computers have difficulty distinguishing the features in
sample input needed to make these applications function, and Mechanical Turk provides an inexpensive and quick method to have humans perform these identification tasks.

Mechanical Turk is centered on the completion of Human Intelligence Tasks, or HITs. A HIT is a “single, self-contained task that a Worker can work on, submit an answer, and collect a reward for completing” (mTurk). These tasks are set up by Requesters, and completed by Workers. Requesters use an API provided by Amazon to set up simple HITs which workers can complete using any web browser, logged into their worker account. These submissions are reviewed upon completion by the Requester, and the Worker is potentially paid, with Amazon receiving a small commission of 10%. (Figure 1) HITs are characterized by the small amount of time required for their completion and the small amount paid to the worker, frequently as low as $0.01.

The Mechanical Turk model places few obligations on either party involved. Workers are able to choose HITs based on a description of the HIT, reward per HIT, number available, and time allotted for completion. A worker can choose to work on any HIT provided he or she meets qualifications set by the requester. However, workers are not obligated to complete tasks, and can stop at any time and work on other tasks. Requesters can set minimum standards for the workers, including experience on the website, number of submissions, and whether these submissions were accepted or rejected by other requesters. Additionally, the requester can choose to make potential workers undergo simple tests to determine their competence in the area of the task. Once the HIT is completed, the requester can review the task and choose to either accept or reject the submission, affecting the worker's public record on the site. Requesters also have the option of giving a bonus to individual workers upon completion of the HIT, in addition to the stated reward for completion.
Mechanical Turk has been used as a knowledge-acquisition backend to generate large datasets for research and software development projects. The accuracy of artificial intelligence systems such as natural language processing and visual recognition systems is improved by training with datasets that correctly match an input with the desired output. Construction of these datasets involves tasks such as matching an image with a set of descriptive words, or matching an audio file with a transcription of that file – tasks that are best performed by humans. Services such as Mechanical Turk provide the ability to divide construction of these datasets up into nearly identical subtasks and distribute them to a large group of workers who are able to work on them independently and in parallel. In certain cases, such as the construction of MIT's LabelMe image annotation dataset, this has allowed the construction of a dataset more quickly and less expensively than traditional methods would have allowed. LabelMe has accumulated over 400,000 images and associated descriptive words since 2005 through the use of Mechanical Turk.
and a similar but non-paying interface (Torralba 2010).

The ease of participation in services such as Mechanical Turk enables widespread use, but also raises concerns about the quality of the task submissions. The anonymous nature of the workers in the crowd makes it theoretically easy to submit erroneous or poor quality work with no lasting effect on the worker's ability to complete future tasks. While Mechanical Turk has methods to try to maintain high quality submissions, such as keeping track of each user's success in completing previous tasks, it is very easy for a worker to disassociate himself from a poor record by creating a new account, or artificially boost his rating by creating, completing, and approving his own HITs (Ipeirotis). However, experimental generation of natural language processing data sets using Mechanical Turk have produced results that compare favorably to those produced by experts, which are typically “extremely expensive in both annotator-hours and financial cost” (Snow 2008). Raw translation and annotation data collected from Mechanical Turk is more abundant but of poorer quality than data produced by linguistic experts for the same tasks. However, through statistical pooling and exclusion of outliers, the non-expert data can be normalized to achieve accuracy very similar to expert produced data.

A study that used Mechanical Turk to generate a range of linguistic data sets found that, on average, only 4 non-expert annotations per example were required to achieve the same accuracy as an expert evaluation. The parallel evaluation of the tasks allowed them to be completed quickly, at a rate of 1724 annotations / hour, and at a low cost, of 875 annotations / dollar (Snow 2008). These results show a vast improvement over the costs of data produced by experts in the field, which have the potential to cost thousands of dollars if conducted in an academic setting using linguists or graduate students.

An additional model that has been used to generate high quality results using Mechanical
Turk involves iterative, rather than parallel, labor. In this model, instead of having different problems solved by different workers in parallel, successive workers do tasks that build each on each other, with one worker's outputs used as the input for the next worker. The TurkIt software developed by Greg Little of MIT's CSAIL builds on the Mechanical Turk interface functions, and automatically generates new HITs based on the results of previous HITs (Little 2009). TurkIt's framework allows for iterative cycles based on two types of tasks: improvement tasks and voting tasks.

In Little's experiment, this cycle was applied to an image description task. A worker was presented with an image and a brief paragraph that described it, and asked to improve the description. After the worker submitted his improvements, another task was generated that presented workers with both the original and the modified description, and asked them to vote on which was better. The cycle then repeated with the more highly rated description as input for another worker's improvement task. Workers on improvement tasks were paid if their corrections were voted to be beneficial, and workers on voting tasks were paid if their vote matched the majority. The study found that the voting mechanism was largely unnecessary because workers usually made beneficial changes. Also, early iterations of the original input text “heavily influenced the structure and tone of the final description” (Little 2009), and that most concrete details about the image were added in early in the process. The results of the iterative method were compared to results generated using the parallel method for the same images, and were found to be of higher quality 82% of the time, despite using the same budget per description.

**Acquisition of Specialized Knowledge: Broadcast Search and InnoCentive**

Micro-crowdsourcing platforms such as Mechanical Turk provide inexpensive methods
for the division of large amounts of unskilled human knowledge accumulation, and have demonstrated that workers are willing to perform simple tasks such as image annotation for rewards as low as a few cents. However, to crowdsource tasks that require specialized knowledge and a significant creative process, such as R&D tasks, scientific problem solving, or software development, a different incentive structure is required. Large-scale collaborative projects require a large amount of concentrated effort to start, and are usually begun by a small group of core contributors, often a single developer.

Work on Wikipedia articles, for example, is typically not equally spread among a wide group of contributors; rather, a single committed individual will begin the article and write most of the initial content for it, to be later refined and expanded upon by other contributors (Wood). In the case of open source software, these initiators of the project generally “become the project owners or maintainers who take on responsibility for project management” (von Hippel 2003), coordinating the ongoing distributed development of the project through email and online message boards. An example is the development of the Linux operating system kernel, which originated from a hobby project written entirely by Linus Torvalds. Torvalds took on the role of head developer and project administrator after soliciting the help of other expert developers, coordinating their efforts and asserting control over the inclusion of the code they wrote. To be successfully crowdsourced, projects such as these that require specialized knowledge require methods of finding experts capable of providing this knowledge.

Karim Lakhani refers to this type of matching of difficult problems with potential outside solvers as “broadcast search.” Broadcast search has the potential to produce novel solutions to unsolved development problems by soliciting the efforts of two groups of individuals (Lakhani 2007). The first group consists of experts in the problem domain who were previously unknown
and unaffiliated with the problem, who perhaps are employed by another company but work in the field the problem is associated with. The second group is made up of individuals with diverse backgrounds, who are experts in fields outside of the problem domain. Experts in fields outside of the problem domain have access to a wider range of methods and design paradigms which they may be able to apply to the problem in unique and innovative ways (Lakhani 2007), solving the problem with a novel application of existing techniques.

Jeppsen and Lakhani have based their classifications of broadcast search on studies of InnoCentive, a company that broadcasts research and development problems to a wide community of users in the form of development contests. Firms with large R&D operations pay InnoCentive to identify which of their unsolved research problems can be broadcast on the InnoCentive website. Members of the InnoCentive community are then able to submit proposed solutions to the problems. The seeker firm reviews the submissions and awards a monetary prize to the proposer of the best solution (Helms 2007). Essential to InnoCentive's implementation of broadcast search is the low barrier to entry into its community – to begin browsing and working on unsolved problems, a solver need only create an account on InnoCentive's site, which can be done in mere minutes.

Ease of participation allows members of both groups described by Lakhani, the unknown experts and the outside experts, to propose solutions to problems that they would not have been able to access otherwise. Indeed, these marginal experts have had disproportionate success in solving the problems broadcast by InnoCentive. Examining winning solutions to a wide variety of research problems, Lakhani and Jeppsen found that a solver's technical and social marginality were positively correlated with his or her probability of success. Female solvers had disproportionately high numbers of winning solutions, as did solvers whose fields of experience
were relatively unrelated to the domain of the problem. The odds of finding a successful solution to a given problem “increases with each additional solver who arrives with a different analytical toolkit and perception of, and angle on, the problem” (Jeppsen).

Applications of Broadcast Search: TopCoder

TopCoder is a software development company that has applied the principle of broadcast search to develop software systems and components. Software development is done by a worldwide community of more than 260,000 freelance programmers in over 200 countries, all of whom compete for monetary prizes by developing software that meets specifications set by TopCoder (Roush). TopCoder acts as a facilitator of a two-way market in a manner similar to InnoCentive. Clients pay TopCoder to design software applications that meets their specifications. TopCoder, in turn, creates a contest that challenges its community members to write software that meets these specifications. Community members who submit the best solutions to these problems, as determined by automated unit testing and human review, receive a prize, which is a share of the client’s payment to TopCoder. TopCoder claims that their competition-based design of software allows the production of higher quality code for less cost, with between 30% and 60% lower costs than traditional outsourcing or in-house production models (TopCoder 2008).

TopCoder has used this contest based system to develop a wide range of software systems, from “something as simple as a Web page all the way up to full-blown enterprise resource planning systems” (Roush). Large scale projects are divided into submodules by TopCoder employees, and then distributed as competitions. In some instances, such as when the client does not have well defined specifications for their system, TopCoder will broadcast every
stage of the design process as small competitions, including conceptualization, specification writing, architecture design, and user interface prototyping. The results of one competition are used as the basis for the next, such as when the winning object oriented design from one competition is used to specify the workings of individual components and algorithms. Once the individual components are completed, they are amalgamated into a finished system through software assembly contests. To ensure that code is robust and maintainable, TopCoder has in-house employees who serve as platform managers and are responsible for directing the division and later integration of the development work. Additionally, when software bugs are reported, TopCoder posts them to an open “BugRace” competition, where community members are paid between $25 and $100 for fixing any unresolved software flaw in any of the systems being currently developed (TopCoder).

Unlike InnoCentive, TopCoder does not employ a “winner-take-all” system in its competitions. Rather, the prize money is split between the first place and second place winners, with the first place winner generally getting about twice as much as the second place (Roush). Prizes are comparatively small, and are rarely more than $3,000 for even the largest projects. Even so, a small group of members of the TopCoder community have been able to earn relatively large sums of money from repeated success in competitions. An example is Tomasz Czajka, who began competing in TopCoder competitions while he was an undergraduate student, and accumulated more than $130,000 over five years (Chen).

However, only the top few individuals in each competition actually receive monetary compensation for their work, and many programmers participate without ever winning and being compensated. Other, non-financial incentives explain the community members’ continued willingness to contribute labor and code even if they are historically unsuccessful in coming in
first or second. TopCoder implements a rating system that reflects a programmer’s success in beating other rated community members, similar to the rating system implemented by the World Chess Federation. Each member has a profile page on the TopCoder website, which displays his or her current rating, overall ranking on the site, and a breakdown of his or her successes and failures. Community members are able to view the profile pages of other members, post in TopCoder forums, and send messages to other community members. This specialized social network provides an extrinsic incentive for community members to continually compete to gain status within the community. Prominent software corporations such as Google and Microsoft have begun to recruit new employees from among the top-ranked members in the TopCoder community. An example is previously mentioned Czajka, who was recruited by Google after winning successive TopCoder open competitions (Chen).

Beyond the extrinsic motivations of prize money, status within the programming community, and career advancement, a possible motivating incentive for participation in TopCoder contests is the intrinsic value of such participation. Intrinsic motivation refers to participation in an activity because of a desire to “seek out novelty and challenges, to extend and exercise one’s capacities, to explore, and to learn” (Ryan and Deci 70). Intrinsic motivation is cited by experts on free and open-source software as a main contributing factor to the unpaid work of software professionals on open-source projects (von Krogh and von Hippel 2003, Lakhani and Wolf 2005). These experts refer to an individual’s total absorption in a coding project as “flow”, an intellectual state characterized by high creativity and intense concentration. Programming competitions, if intellectually stimulating to the participants, have the capability to induce this enjoyable state of flow in the programmers. Participation in coding contests also has educational value, and the “enormous.. technical learning opportunities” (von Krogh and von
Hippel 2003) promise to increase the programmer’s future returns on knowledge acquired through the contests. Indeed, the first competitions hosted by TopCoder in the early 2000s were Collegiate Competitions open only to full-time university students, who still make up a large part of the community (Meloan). TopCoder participants may find problems that provide sufficient intellectual stimulation and educational value to motivate their involvement even without the promise of financial compensation or rating increase.

Bourdreau, Lacetera, and Lakhani (2008) conducted research on TopCoder competitions to determine if increasing the number of participants in a given coding competition increases the quality of the code produced in the competition, as measured by the code’s performance rankings. TopCoder provides a competitive atmosphere by placing groups of contestants in virtual rooms of about fifteen people during competitions, where they are able to see the rankings of all competitors in the room and communicate with one another. Bourdeau reports that most programmers don’t use the communication tools to share ideas and collaborate on the problems, but rather to “trash talk” the other members in the room. This communication, combined with the live rankings of members as they submit their entries over the short duration of each competition, contributes to a game-like competitive atmosphere. The researchers detail two effects of increased competition: the competition effect, which refers to the lowering of average individual performance scores as competition increases; and the parallel search effect, which refers to the increase in the maximum performance score as competition increases. Adding more competitors to a competition produces a paradoxical effect: average code quality decreases, but the quality of the best entry increases. The researchers also draw attention to the complexity of the programming tasks studied. More complex programming tasks made the competition effect weaker and the parallel search effect stronger; that is, average individual performance did not
degrade as much for complex tasks as competition increased, and maximum performance increased even more for complex tasks as competition increased. This research suggests that the competition model is particularly suited to development of complex code, where problems lie in multiple domains, and effective solutions rely on greater creativity and clever coding techniques (Bourdeau et al. 2008).

**Iterative Competitive Design: MathWorks Competitions**

MathWorks, the company behind the popular MATLAB mathematical and programming software, hosts semi-annual programming competitions that are reminiscent of the TopCoder competitions, but differ in several crucial aspects. The competitions are centered around a class of mathematical problems known as NP-complete problems, which are difficult algorithmic problems that are easily testable but difficult to code efficient solutions for. Solutions to MathWorks problems, like the algorithm problems presented in TopCoder, must be efficient, robust, and capable of handling a wide range of inputs. MathWorks competitions, like TopCoder competitions, have low barriers to entry, and are open to anyone with an account on the MathWorks website and a copy of the MATLAB software, which all entries must be programmed in. However, unlike TopCoder competitions, where all code submissions are private until the code submission period has ended, MathWorks competitions provide a window of time when all code submitted to the contest is viewable by all contestants. This allows contestants to optimize the submissions of others, and resubmit the improved code as their own in an iterative design process (Gulley and Lakhani 3). Also, MathWorks does not provide a monetary prize to the competition’s winner. Although the top competitors receive some small merchandise such as a t-shirt, the participants often put in hours of programming labor into the week-long
competitions to be rewarded only with their name and a brief interview posted on the MathWorks website. Despite the unpaid nature of the work, programmers tend to be very involved in the contest process, and many are fiercely competitive as they seek to gain the top spot on the contest leaderboard, if only temporarily (Gulley and Lakhani).

The unique nature of code reuse and “collaborative competition” in the MathWorks competitions, along with the objective measurement of code performance provided by algorithmic speed and unit testing, provides an opportunity to study the effects of iterative competition on code performance. Gulley and Lakhani studied the relationship between the structure of code and its success and reuse by the community. They found that more complex code was likely to achieve a high rating and be reused by the community. Additionally, code that didn’t conform to accepted coding standards (i.e. the programmer wrote the code in a convoluted manner) was more likely to achieve a high rating, but was less likely to be reused by other members of the community (Gulley and Lakhani). Some programmers in the competitions gained reputations as good “hackers”, programmers able to take non-conforming shortcuts that boosted algorithm performance. Other competitors emerged as “talent scouts”, able to identify the good hackers and combine their submissions in a manner that further increased performance. In one contest, only about 20% of the winning entry was actually written by the entry’s submitter. The remainder of the code could be traced back to 30 other previous submissions by other programmers, an impressive number considering that most contests have only between 100 and 150 participants (Gulley and Lakhani). However, by and large, the non-conforming code written by the “hackers” is less adopted, perhaps because of other members’ difficulty in understanding it, or its fragile nature and the errors that could be easily introduced when it is modified. In the end, the best code produced was highly efficient in producing answers to the
difficult contest problems. MathWorks is hesitant to apply this style of design to their own commercial products because of the intellectual property rights surrounding code iteratively developed by dozens of non-employees and the necessarily open source nature of such a product, but has stated that the code produced by these contests is far superior to anything that MathWorks could have produced in-house (Gulley and Lakhani). The design process is also very inexpensive for MathWorks, which only maintains the contest mechanisms and supplies the winners with nominal prizes.

**Conclusion**

Online frameworks for software collaboration allow the production of cost-effective, robust software that can achieve the same level of algorithmic performance as software produced by traditional in-house means. The Internet’s lowering of transaction costs allow the coordination of both the collaborative and competitive efforts of globally distributed programmers, who are often motivated by intrinsic and non-financial extrinsic factors. In such settings, competitive design is well-suited to the development of complex programs in a cost effective manner. Building on the framework of the open source software movement, the collective talents of distributed communities of software developers can be utilized to produce and improve upon software designs and code.
Sources


