CROWDY
A FRAMEWORK FOR SUPPORTING SOCIO-TECHNICAL SOFTWARE ECOSYSTEMS WITH STREAM-BASED HUMAN COMPUTATION

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Mert Emin Kalender

August, 2014

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Bedir Tekinerdoğan(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Buğra Gedik

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Halit Oğuztüzün

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

CROWDY
A FRAMEWORK FOR SUPPORTING SOCIO-TECHNICAL SOFTWARE ECOSYSTEMS WITH STREAM-BASED HUMAN COMPUTATION

Mert Emin Kalender
M.S. in Computer Engineering
Supervisor: Asst. Prof. Dr. Bedir Tekinerdoğan
August, 2014

The scale of collaboration between people and computers has expanded leading to new era of computation called crowdsourcing. A variety of problems can be solved with this new approach by employing people to complete tasks that cannot be computerized. However, the existing approaches are focused on simplicity and independency of tasks that fall short to solve complex and sophisticated problems. We present Crowdy, a general-purpose and extensible crowdsourcing platform that lets users perform computations to solve complex problems using both computers and human workers. The platform is developed based on the stream-processing paradigm in which operators execute on the continuos stream of data elements. The proposed architecture provides a standard toolkit of operators for computation and configuration support to control and coordinate resources. There is no rigid structure or requirement that could limit the problem-set, which can be solved with the stream-based approach. The stream-based human-computation approach is implemented and verified over different scenarios. Results show that sophisticated problems can be easily solved without significant amount of work for implementation. Also possible improvements are discussed and identified that is a promising future work for the existing work.

Keywords: Crowdsourcing, human computation, stream processing.

ÖZET

CROWDY
SOSYAL VE TEKNİK YAZILIM EKOSİSTEMLERİNİ DİNAMİK KİTLE KAYNAK İLE DESTEKLEYEN UYGULAMA

Mert Emin Kalender
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Dr. Bedir Tekinerdoğan
Ağustos, 2014

İnsanlar ve yazılım bileşenleri arasındaki işbirliği gelişerek kitle kaynağın ortaya çıkmasını sağlamıştır. Kitle kaynak kullanılarak yazılım tarafından çözülemeyen veya çözülmesi zor birçok sorun insanlar aracılığı ile çözülmüş ve bir sonuca ulaşılmıştır. Fakat günümüzdeki kitle kaynak odaklı yaklaşımlar ve çözüm süreçleri yapılan işlerin basitliğine ve birbirlerinden bağımsız olmalarına ağırlık vermektedir. Bu sebepten dolayı zor ve çok yönlü sorunların çözümü mevcut yaklaşımlarla mümkün değildir. Bu çalışmada kullanıcıların sorun çözümü konusunda hem insanları hem de yazılım bileşenlerini kullanabilecekleri, genel amaçlı ve geliştirilebilir bir yaklaşım ve bu yaklaşımın uygulandığı bir altyapı sunulmaktadır. Yaklaşım küçük işlemcilerin sürekli olarak akan veriler üzerinde çalışması mantığına dayanmaktadır. Sunulan altyapı bünyesinden barındırdığı temel işlemciler sayesinde işlem kaynakları arasındaki kontrol ve eşgüdümü kolaylıkla sağlamaya elverişli şekilde tasarlanmıştır. Sunulan yaklaşım kısıtlı bir sorun listesini hedeflememektedir ve kullanıcılar açısından herhangi bir kısıtlama getirmemektedir. Çeşitli örnekler yapılan incelemeler sunulan yaklaşımın sorunları kayda değer bir iş yükü getirmeden çözülebileceğini göstermiştir. Ayrıca, çeşitli iyileştirme önerileri de tartışılmış ve bazıları gelecek çalışmalara eklenmek üzere belirlenmiştir.

Anahtar sözcükler: Kitle kaynak.

Acknowledgement

First and foremost I would like to express my sincere gratitude to my advisor, Asst. Prof. Dr. Bedir Tekinerdoğan, who let me explore my research ideas and has supported me throughout my thesis with his guidance, knowledge and patience. This work would not have been completed without him, to whom I am greatly indebted.

Besides my advisor, I would like to thank the rest of my thesis committee: Asst. Prof. Dr. Buğra Gedik and Assoc. Prof. Halit Oğuztüzün for their encouragement and insightful comments.

I am thankful to all the faculty at Bilkent University, especially in the Computer Engineering department. They were always there to teach, discuss and support any matter. I have been blessed with a friendly and sophisticated group of faculty members. My research would not have been possible without their help.

I would also like to thank the current and former staff at Bilkent University for their various forms of support during my study. Among them I express my warm thanks to Nimet Kaya for being one of a kind.

I thank to my fellow classmates and colleagues. I am glad to have interacted with many on various occasions.

I am grateful to my friends for their support and encouragement throughout. I have to give a special mention for the support given by Mehmet Volkan Kaşıkçı and Abdullah Başar Akbay.

Finally, I would like to thank God, whose many blessings have made me who I am today. A year's worth of effort, frustration and achievement is summarized in the following document.

Contents

1 Introduction **1**

1.1 Problem Statement 2

1.2 Purpose 4

1.3 Thesis Plan 5

2 Crowd Computing **7**

2.1 Preliminaries 7

2.1.1 Tasks 9

2.1.2 Requesters 12

2.1.3 Workers 13

2.2 Building Blocks of a Platform 13

2.2.1 Workflow Design 14

2.2.2 Task Assignment 15

2.2.3 Quality Control 16

3 Platform **18**

3.1 Software Architecture of the Platform 19
3.1.1 Views and Beyond Approach 20
3.2 Application Development on the Platform 26
3.2.1 Operator 29
3.2.2 Flow 38
3.3 Flow Composition 38
3.3.1 Source operators 39
3.3.2 Processing operators 40
3.3.3 Relational operators 40
3.3.4 Utility operators 41

4 Tool **42**

5 Case Studies **52**

5.1 Verifying Business Information 52
5.2 Translation 55
5.2.1 Naive Approach 56
5.2.2 A More Sophisticated Approach 58
5.2.3 Final Approach 59

6 Related Work **61**

6.1 Crowdsourcing Platforms 61

6.1.1 Jabberwocky 62
6.1.2 WeFlow 63
6.1.3 TurKit 64
6.1.4 CrowdLang 65
6.1.5 AutoMan 65
6.1.6 Turkomatic 66
6.1.7 CrowdForge 67
6.1.8 CrowdWeaver 68
6.2 Other Studies 70
6.2.1 Soylent 70
6.2.2 Qurk 70
6.2.3 Mobi 71
6.2.4 CrowdSpace 71
6.2.5 Human Architecture 72

7 Discussion **73**

8 Conclusion **75**

8.1 Future Work 76
8.1.1 Extending the Operator Set 76
8.1.2 Improvements to Existing Operators 77
8.1.3 More Quality Control 78

List of Figures

2.1 The frequency of crowdsourcing keyword in academic papers. . . . 8

2.2 Task asking people to find address of a company 10

2.3 Task asking people to distinguish sentiment in a text snippet . . . 11

3.1 Structure of the client module of the platform. 20

3.2 Structure of the application server module of the platform. 21

3.3 Architecture of the platform. 22

3.4 Client-side architecture. 23

3.5 Server-side architecture. 23

3.6 Detailed architecture in terms of components and connectors. . . . 25

3.7 Architecture of the platform. 26

3.8 Crowdy application correspondence to Pipe-and-Filter style. . . . 27

3.9 Metamodel for a Crowdy application. 27

3.10 A sample, minimal *Crowdy* application. 28

3.11 Base operator representation. 29

3.12 Source operator representation. 30

3.13 Sink operator representation. 33

3.14 Processing operator representation. 33

3.15 Selection operator representation. 34

3.16 Sort operator representation. 35

3.17 Enrich operator representation. 36

3.18 Split operator representation. 37

3.19 Union operator representation. 38

4.1 Flow composition window. 43

4.2 An source operator added to flow. 44

4.3 Configuration window for source manual operator. 45

4.4 Configuration window for source manual operator filled with sample information. 46

4.5 Two operators connected via flow. 47

4.6 Configuration window for human processing operator filled with sample information. 48

4.7 Human task preview window for human processing operator. . . . 48

4.8 A minimal application with source, processing and sink operators. 49

4.9 Configuration window for sink operator. 49

4.10 Validation windows listing warnings and errors. 50

4.11 Configuration window for sink operator after validation. 50

4.12 Successful validation window with submit link. 51

4.13 Window to report a bug. 51

5.1 Crowdy application to correct business addresses. 53

5.2 Sample list of companies and their information. 53

5.3 Human task that is generated for human workers. 54

5.4 Human task that is generated for human workers (updated). . . . 55

5.5 Another approach to correct business addresses. 56

5.6 Some part of Rumi's Poem "Etme". 56

5.7 Crowdy application to translate a text. 57

5.8 Human task that is generated for human workers for translation. . 57

5.9 Crowdy application to translate a text. 58

5.10 Some part of Rumi's Poem "Etme". 59

5.11 Crowdy application to translate a text. 59

5.12 Human task that is generated for human workers to check quality of translation. 60

List of Tables

3.1 List of inputs and options 32

6.1 Comparison of existing crowdsourcing platforms. 69

Chapter 1

Introduction

The growing software systems are characterized by asynchrony, heterogeneity and inherent loose coupling promoting system of systems as a natural design abstraction. The new system concept goes beyond the size of current definition by several measures such as number of people the system employed for different purposes; number of connections and interdependencies among components; number of hardware elements; amount of data stored, accessed, manipulated, and refined and number of lines of code [1]. These requirements lean towards a decentralized and dynamic structure that is formed by various systems interacting in complex ways.

Therefore, software system becomes an ecosystem in which components supported by a common platform operate through exchange of information, resources and artifacts and contribute to the overall service that system tries to provide [2]. In fact, the components that are fundamental to system functionalities are not only software components, but there are now components whose functionality is operated by human beings. People become not only users, but also an integral part of the system providing content and computation, and the overall behavior [1].

Human involvement not only makes the system gain a social characteristic in

addition to it's technical features, it also gives ability to solve numerous problems, including the ones requiring human intelligence. In that sense, the scale and variety of components involved within the system increases significantly, and homogeneity of components cases respectively. The difference between the roles concerning system components and humans (user, developer) becomes less distinct. Humans take an essential part of the system in collaboration with software components.

The scale of collaboration of creative and cognitive people with number-crunching computer systems has expanded from small or medium size to internet-scale [3] leading to new era of computation. Although this collaboration has appeared under many names such as human computation, collective intelligence, social computing, global brain etc, crowdsourcing is the main term that is being used to refer to human and computer collaboration.

Crowdsourcing as the new and powerful mechanism of computation has become compelling to accomplish work online [4]. Over the past decade, numerous crowdsourcing systems have appeared on the Web (Threadless, iStockphoto, InnoCentive etc). Such systems enable excessive collaboration of people have provided solutions to the problems and tasks that are trivial for humans, which cannot be easily completed by computers or computerized. Hundreds of thousands of people have worked on various tasks including deciphering scanned text (recaptcha.net), discovering new galaxies (galaxyzoo.org), seeking missing people (helpfindjim.com), solving research problems (Innocentive), designing t-shirts (Threadless). Even Wikipedia and Linux can be viewed as crowdsourcing systems from a point of view that conceives crowdsourcing as explicit collaboration of users to build a long-lasting and beneficial artefact [5].

1.1 Problem Statement

Although current crowdsourcing systems allow a variety of tasks to be completed by people, the tasks requested for completion are typically simple. Tasks, often

described as micro-tasks, have the two following fundamental characteristics:

Difficulty. Tasks are narrowly focused, low-complex and require little expertise and cognitive effort to complete (taking a couple of seconds to a few minutes).

Dependency. Tasks assigned to humans are independent of each other. The current state of one job has no effect on the other. The result of one job cannot be input to the other to create some information flow.

In that sense, simplicity makes the division and distribution of tasks among individuals easy [6], and independency enables parallelizing and bulk-processing tasks. However, solving more complex and sophisticated problems requires effective and efficient coordination of computation sources (human or software) rather than creating and listing a series of micro-tasks to-be-completed.

Recently detailed analysis on current mechanisms based on foundations of crowdsourcing reveal the necessity of a more sophisticated problem-solving paradigm [7]. Researchers explicitly state the need for a new generic platform with the ability to tackle advanced problems. Kittur et al. [7] suggest researchers to form new concepts of crowd work beyond the simple, independent and deskilled tasks. Based on the fact that complex work cannot be accomplished via existing simple and parallel approaches, the authors state the requirement for a platform to design multi-stage workflows to complete complex tasks, which can be decomposed into smaller subtasks, by appropriate groups of workers selected through a set of constraints.

In another piece of work, Bernstein et al. [8] regard all the people and computers as constituting a *global brain*. Authors indicate the need for new powerful programming metaphors that can more accurately demonstrate the way people and computer work in collaboration. These metaphors are expected to solve dependent sections of more complex problems by decompositions and management of interdependencies. Further, the specification of task sequence and information flow are expected to enable deliberate collaboration over solutions.

However, recent research only partially addresses these challenges by providing programming frameworks and models [4, 9, 10, 11, 12, 13, 14, 15] for massively parallel human computation, limited-scope and ability user interfaces [16, 17, 18, 19], concepts for planning [20], analysis of collaboration [21]. These studies fail to tackle challenges of crowdsourcing due to following reasons:

- having rigid structure and requirements due to the (programming) concepts and libraries that they are based on
- being only applicable to a small and bounded problem-set
- requiring a significant amount of work in order to implement and integrate a crowdsourcing solution to solve a problem.

Further, human workers are often regarded as homogeneous and interchangeable due to the issues of scalability and availability in existing mechanisms [9]. However, people in a crowd have different skills, and can perform different roles based on their interests and expertise [6]. Current services are created without considering the availability and preferences of people, constraints and relationships, and the support of dynamic collaborations [22]. Thus, human involvement in current mechanisms should be rethought due to limited support for collaboration and ignorance of collaboration patterns in problem-solving [23] over general-purpose infrastructures that can more accurately reflect the collaboration of people and computers [8, 24].

Nevertheless, the development of more generic crowdsourcing platforms along with new applications and structures are expected by the research community [5].

1.2 Purpose

This study aims to solve the previously mentioned issues associated with crowdsourcing platforms and tackle challenges in crowdsourcing. In order to achieve this, the existing frameworks and platforms are identified, analyzed and discussed

thoroughly. Based on the data and feedback extracted out of the related studies, a new general-purpose and extensible framework is proposed. The framework is implemented into a tool and evaluated through real-world case studies.

The proposed platform, called *Crowdy*, is developed to support software ecosystems via stream-based human computation. *Crowdy* can be used to design and develop crowdsourcing applications for effective and efficient collaboration of human and software components. The platform

- enables users to perform computations to solve complex problems
- has no rigid structure or requirements
- is not limited to a specific problem-set or aspect of crowdsourcing

The platform consists of an application editor, a runtime environment and computation resources. Users design applications by simply creating and connecting operators together. These applications are submitted to runtime environment. The runtime environment executes applications by creating processes. A process is performed via corresponding computation resources, which can be either people or software. In the case of human computation existing crowdsourcing services are used. Otherwise, computers are utilized.

This platform is concentrated on providing mechanisms that can be used to decompose the implementation of an application into a set of components as a close representation of the real-world problem. The main characteristic of this work is to show how sophisticated problems can be accomplished cleanly and easily relying on component-based model.

1.3 Thesis Plan

The remainder of the work continues as follows. In Chapter 2, background information on crowdsourcing is given. Chapter 3 provides the details of the proposed

platform and concepts used in this work. In Chapter 4, the tool developed over the platform is demonstrated. Various motivating scenarios are described and developed using the proposed platform in Chapter 5. Related studies are examined in Chapter 6. In Chapter 7, a brief discussion of future work is given. A final chapter concludes this study.

Chapter 2

Crowd Computing

In the following, first the preliminary information about crowdsourcing concepts is summarized, since crowdsourcing is a relatively area of computation. The building blocks of a crowdsourcing platform are also described.

2.1 Preliminaries

Collaboration of creative and cognitive people with number-crunching computer systems have appeared under many names such as crowdsourcing, human computation, collective intelligence, social computing, global brain etc, for which you can find detailed studies on classification of systems and ideas in [25, 26] collected under distributed human computation term.

The term *crowdsourcing*, which is the main consideration in this body of work, is first coined by Jeff Howe in the June 2006 issue of Wired magazine [27] as an alternative to the traditional, in-house approaches focusing on assigning tasks to employees in the company for solving problems. Crowdsourcing describes a new, mainly web-based business model that exploits collaboration of individuals in a distributed network through an open call for proposals. The term is described by Howe as follows:

> Simply defined, crowdsourcing represents the act of a company or institution taking a function once performed by employees and outsourcing it to an undefined (and generally large) network of people in the form of an open call. This can take the form of peer-production (when the job is performed collaboratively), but is also often undertaken by sole individuals. The crucial prerequisite is the use of the open call format and the large network of potential laborers. [28]

Although there is a long list of terms (collective intelligence, social computing, human computation, global brain etc.), in which some of them are new and some others are old, the use of term "crowdsourcing" in the academia (demonstrated in Figure 2.1) indicates that research on this domain is promisingly increasing.

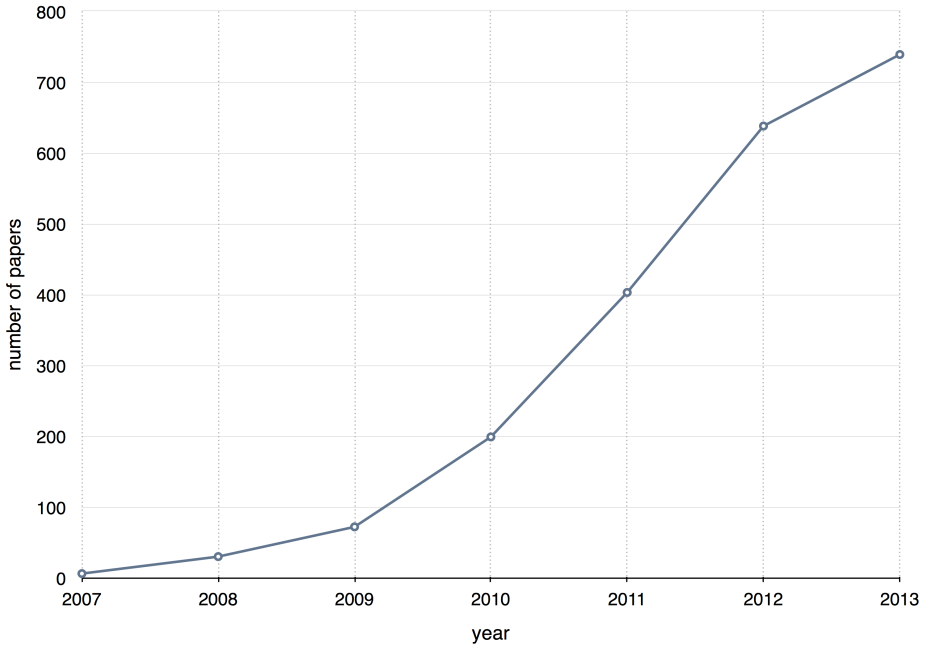

Figure 2.1: The frequency of crowdsourcing keyword in academic papers.[1]

Researchers have been exploring different approaches to employ crowd of people in solving various problems. From task creation to quality control there has been a lot of research in crowdsourcing. In most of these studies, Amazon's Mechanical Turk (MTurk) is the main consideration on crowdsourcing where new proposals are implemented and executed on this platform.

MTurk is a general purpose crowdsourcing platform recruiting large numbers of people to complete diverse jobs. The platform acts like an intermediary between employers (known as requesters) and employees (workers or turkers) for various-sized and difficulty assignments. Assignments on MTurk range from labeling images with keywords, transcribing an audio snippet, finding some piece of information on the Web. Requesters submit jobs, which are called Human Intelligent Tasks or HITs in MTurk parlance, as HTML forms. Respectively workers, who are the crowd of users, (called Turkers on MTurk) perform or complete these jobs by inputting their answers and receiving a small payment in return. This actual platform and other example systems listed above present the potential to accomplish work in different areas within less time and money required by traditional methods [24, 16]. The platform has become successful for tackling the problems that cannot be solved by computers and subject to numerous research studies.

Similar to other academic studies, MTurk is considered in this work to discuss the common terms and form a preliminary list of concepts. In the following, the fundamental concepts of crowdsourcing are explained.

2.1.1 Tasks

Task is a piece of work to be done by the crowd. The terms task, micro-task and job are interchangeably used. The term Human Intelligent Task or HIT is commonly used as well. A task is often expressed over an HTML form in which there are three different input types: single selection (radio buttons), multiple selection (checkboxes) and free text (text areas). For single and multiple selection one or more items can be selected from a list of possible options. Considering free text types workers are supposed to enter a textual response that can be paragraph(s) or sentence(s) or number(s).

In addition to the labels attached to input forms, task has a short description that provides the instructions and keywords, which will help workers search for

[1]The statistics are gathered through ACM Digital Library.

specific tasks. Further, number of assignments (copies) that requester wants completed per task can be set. Additional copies of the same task allows parallel processing. In that case, the system is supposed to ensure that the parallel workers are unique (i.e., no single worker complete the same task more than once).

A task is generally simple requiring small amount of time and expertise to complete. In the following, sample tasks from MTurk are presented.

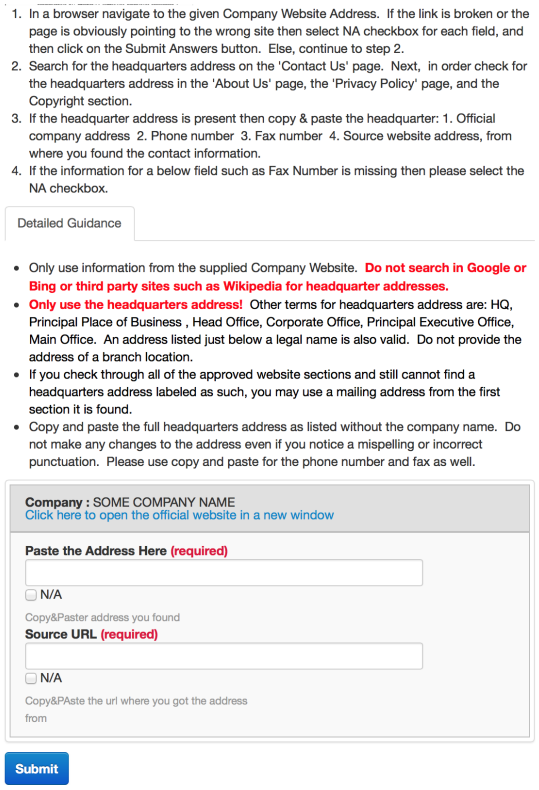

Figure 2.2: Task asking people to find address of a company

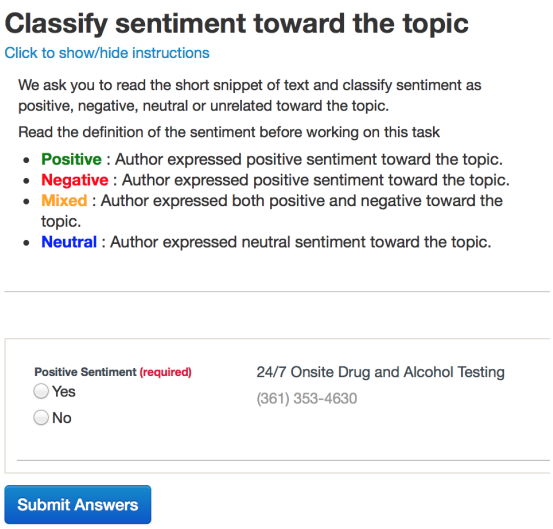

Figure 2.3: Task asking people to distinguish sentiment in a text snippet

Tasks can be grouped into task groups. Task groups are for the tasks that share similar qualities such as tasks to tag images of nature and people or tasks asking for translation of a text snippet from a language to another.

2.1.1.1 Time

Each task is associated with a time value that determines the maximum time allotted per assignment, which can be set by requester. A task should be completed within the time range associated with the task, otherwise task completion fails and task remains uncompleted until some worker completes it within given time.

Time it takes to complete a task is different for each work item. It is possible to have a task that expected to be completed within seconds. However, it is also possible to have tasks that can take hours. Nearly 20% of tasks takes less than 1-hour, and more than half of tasks does not take more than 16-hours [29].

2.1.1.2 Payment

Compensation or reward for completing tasks range from a single penny to dollars. An analysis of MTurk showed that 90% of tasks pays $0.10 or less [29].

2.1.1.3 Acceptance

Once a worker completes and submits an assignment, the requester is notified. The requester has an option to either accept or reject the completed assignment. Acceptance of an assignment, which indicates the work done is satisfactory, makes the worker who completed it get paid, on the other hand rejection withholds payment for which the reasoning may or may not be provided by the requester. Another option is automatic approval, which is the case when requester does not review work after a some time that can be set by the requester.

2.1.1.4 Expiration

Tasks have a lifetime limit that decides the amount of time that a particular task remains in the listings. Lifetime can be set by requester. After a task reaches end of its lifetime, it is automatically pulled from the listings.

Another type of expiration can happen while a worker is operating on an assignment. When workers accept an assignment, the assignment is reserved for them making no other worker to accept it. The reservation is for particular piece of work (in this case assignment) and time-limited. If worker does not submit the completed assignment in allotted time, then reservation is cancelled and assignment is made available for others again.

2.1.2 Requesters

Requesters are the employers who post tasks, obtain results and pay workers. Requesters are expected to design tasks with all the details (description, instructions,

question types, payment, expiration settings etc.) After completed assignments are submitted, requesters can review them by accepting or rejecting, and collect the completed work.

These operations can be done via user interface or application programming interface (API) if there is one.

2.1.3 Workers

Workers are the online users or someone from the crowd who work on and complete assignments in exchange for a small payment. On some platforms (currently not available on MTurk), detailed information about workers are gathered and kept in a database. This information can be later utilized by requesters while associating specific constraints to the assignments such as limiting age to some range for a specific task.

2.2 Building Blocks of a Platform

Kittur et al. provides a detailed description for the future crowdsourcing platform in [7]. Based on this description building blocks of a crowdsourcing platform are defined in the following.

A crowdsourcing platform is a platform to manage tasks and workers in the process of solving problems through multi-stage workflows. Platform should enable decomposition of complex tasks into subtasks and assign them appropriate group of workers. Workers are formed by people with different skills and expertise. The motivation of workers is guaranteed through various approaches such as reputation or payment. Quality assurance is required to ensure the output of a task is high quality and contributes to solution.

Regarding this definition, a crowdsourcing platform consists of three basic elements:

2.2.1 Workflow Design

The existing problem solving approach is simple and depends on parallel bulk processing. There is no real sense of solution or workflow design. Problems are solved in such a way that independent simple tasks are created by requesters, and they solved by workers and each independent solution is processed individually by requesters to come up with final solution. Complex and sophisticated problems cannot be solved by the current simple parallel approach. The reason is that these problems have interdependent portions with changing space, time and skill-set requirements. On the other hand, current approaches undertake problems through bulk processing of independent and simple tasks that require no specific expertise. Therefore, a more appropriate problem solving approach is required.

The need for a more advanced approach is mentioned in [7]. Authors indicate $workflows$ in the sense of decomposing problems into smaller chunks, coordinating dependencies among these chunks and combining results from them. In that sense, workflow design refers to the development of applications to solve complex problems. It considers design and management of computation resources to solve the problem.

Researchers often apply existing programming paradigms to problem specification and workflow design. MapReduce [4, 9], Divide-and-Conquer [14], Iterative [11], Workflow [10] are some of these approaches taken for organizing and managing complex workflows. Although these paradigms perfectly fit to some problems, there are other problems not suited well to those.

In terms of crowdsourcing, workflow could involve different scale of operations that can only be solved by diverse set of actors. A workflow may tackle highly dependent tasks (e.g., translating a poem and assuring its quality) or massively parallel tasks (e.g., finding some piece of information on the Web). Therefore, the workflow design has diverse space requirements. For example, MapReduce is great for dividing a problem into different chunks and solving those chunks separately and finally merging small chunks of solutions into one. However, some problems may require iterations and each iteration or solution chunks may affect

the result of the following solution attempts. Translation is an example that cannot be solved by either MapReduce or Divide-and-Conquer or Iterative approaches. The problem has interdependent problem pieces that depend on each other and iteration is needed to achieve to final solution.

2.2.2 Task Assignment

In the context of crowd work, coordination of limited resources is prominent. Not all problems are simple enough to be solved by simple human tasks such as finding an address or matching a tag and an image. Complex and sophisticated problems requires various types of tasks, which can be either solved by human beings or software programs, to be completed for the final solution. Thus, coordination and collaboration of different computation resources become essential. Crowdsourcing problems have pieces that cannot be computerized, but require human intelligence. In addition, there may be other problems that require both human and software resources.

Although availability and assignment of software resources can be managed by certain algorithms, the case for human resources is rather unpredictable and hard to solve. Human resources are formed by workers and the availability of a worker cannot be predicted or known at any time. In fact, human resources are not homogenous like computers. They are rather heterogeneous. Not like computing resources human resources are different in terms of homogeneity. Each human being in the crowd has a specific set of skills. A worker may have developed good skill set to transcribing audio into text, but at the same time it is not guaranteed that she is good at translating English into Chinese or vice versa. Therefore, there may be a situation in which a task couldn't be assigned to any available workers due to mismatch between task's requirements and skill-set of available workers. Ideally workers are employed with tasks that fit their area of expertise. In practice, requesters may disregard this constraint, but results would not be useful at all.

One important point is that existing systems are misleading by assuming

human crowd is homogeneous. Task assignment needs to be redefined by demonstrating aspects of human resources.

2.2.3 Quality Control

Quality control is a big challenge for crowdsourcing, since low quality work is common. Although crowdsourcing provides high throughput with low costs, the task completion can be highly subjective and makes it susceptible to quality control issues. Workers tend to minimize the amount of effort in exchange of payment. Cheating and gaming the system is often expected in crowdsourcing. In fact, it is shown that low quality submissions can compromise up to one third of all submissions [17].

Badly designed solution proposals, unclear instructions and task definitions, and workers' misbehavior can lead to faulty solutions too. In the survey conducted in [7], workers indicated that a solution that is not properly designed may cause misunderstandings, thus inaccurate solutions. They also mentioned that crowds may intentionally work on a piece of work to cause a flawed solution to trick the system. As a result, researchers have started to investigate several ways of detecting and correcting for low quality work.

Visualization of workflow is one of the methods employed for which directed graphs are used to show the organization of crowdsourcing tasks, allowing endusers to better understand the problem and proposed solution design [14, 15].

Inserting 'gold standard' questions into an assignment by which workers who answer them incorrectly can be filtered out or given feedback [30]. However, writing validation questions create extra burden to requesters and may not be applied to all types of tasks.

Majority voting to identify good submissions is proposed as another option [30, 17], but this technique can be affected by majority (especially when the possible options are constrained) or failed in situations where there are no answers in common such as creative or generative work [19].

Systems (including MTurk) often do not apply any of these quality control approaches, but provide other ways to achieve good workers and discourage bad ones. Currently each worker on MTurk has an acceptance rate that is updated after requester's review on completed assignments. However, this feature does not differentiate one type of task from the other in terms of effect on acceptance rates and that makes it a limited utility. A worker who is skilled in audio transcription would probably have high accuracy rating in related tasks. However, there is no way to reason that this worker can also perform English-to-Turkish translation tasks. Even worse is that workers who pick and complete easy tasks would probably have higher accuracy rates than the ones who choose to perform tasks that require time and expertise [13].

Requesters can utilize acceptance rates by assigning a low limit of them to assignments, which allow only workers with acceptance rates above-limit to accept assignments.

Chapter 3

Platform

Crowdsourcing based on simple, independent and deskilled list of tasks basically limits the scope and complexity of problems can be tackled. Independent tasks that are narrowly focused and low-complex cannot be easily utilized to solve complex problems. Although simplicity and interdependency enable easily distributed and parallelized processing, complex and sophisticated problems require efficient and effective coordination in which problem itself is the decomposed into multiple stages. Each stage requires allocation of appropriate resources, in addition to the management of interdependencies among stages.

The need for new powerful programming paradigm that can handle the advanced problem-solving is clearly stated [5, 7, 8]. The new paradigm or framework is required to be able to

- tackle advanced problems with no rigid limitations
- decompose problems into smaller pieces (or tasks)
- manage dependencies among different tasks
- allocate appropriate resources to each task
- reflect the skill-set of people and computers while solving tasks.

Crowdy is an extensible, general-purpose crowdsourcing platform to solve complex problems. The platform is developed over the fundamentals of stream processing paradigm for which a series of operations are applied to the continuous stream of data elements via operators [31].

Crowdy is an operator-centric platform. Using this platform, a requester with no requirement of a programming background can quickly translate a complex problem into a crowdsourcing application by simply selecting operators and connecting these operators together. As a result of *Crowdy*'s focus on operators, requesters can design applications by selecting right set of building blocks that are necessary to solve their problem, and customizing these blocks particular to the computation to-be-conducted.

Crowdy embodies several features:

- A standard toolkit of operators that can both human and software resources (human or software) to accomplish various tasks
- Configuration support to control and coordinate resource utilization
- Customizable collaborations over parameterization
- Application runtime interface

3.1 Software Architecture of the Platform

Crowdy platform is implemented as a REST [32] architecture. Applications are developed, configured and validated on the client-side. These applications are submitted to server-side via an application programming interface (API) over HTTP. The execution happens on the server-side and results are kept in a database server.

3.1.1 Views and Beyond Approach

The platform architecture design is explained using Views and Beyond approach [33]. Views and Beyond approach is useful for identification and documentation of design decisions made. The approach is a collection of techniques for preliminary documentation for an architecture to find out stakeholders need, give information about design decisions, check if the requirements are satisfied and package the necessary information together. The software architecture of the platform is described in terms of how it is structured, how the runtime behaves and interacts, and how it relates to non software structures under module, component-and-connector and allocation views in the following.

3.1.1.1 Module Views

The platform has two main modules: client and server. These modules interact with each other based on API defined by server module through HTTP.

Crowdy applications are developed on the client-side, which has two main modules: *workflow editor* and *static analyzer* as shown in Figure 3.1.

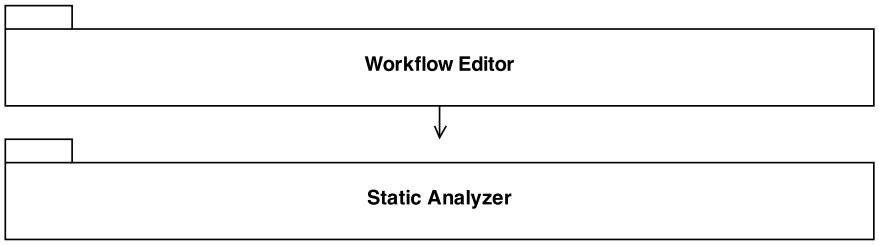

Figure 3.1: Structure of the client module of the platform.

The applications developed on the client-side are executed on the server-side. This part consists of three modules: *API*, *virtual machine* and *service adapters* that is presented in Figure 3.2.

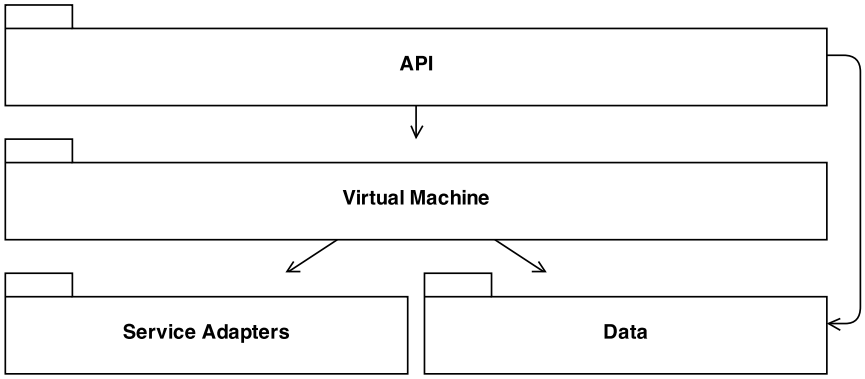

Figure 3.2: Structure of the application server module of the platform.

3.1.1.2 Component-and-connector Views

Component-and-connector (C&C) views are used to document presence of runtime elements and their interactions.

Figure 3.3 illustrates the primary presentation of the system's runtime architecture. It is showing a picture of the system as it appears at runtime. A set of clients can interact with the application server, embodying a client-server style. The client components communicate with application server components via the API defined by the server. Client components can achieve a set of operations such as creating a new application, getting results for an existing application over that API.

The system contains a shared repository of *tasks* in *database server* accessed by the application server. The *task* database is the list of action items per application. New tasks can be created for new applications or a list of uncompleted tasks can be completed with an API call from the application server. Thus, the connection between the application server and the task database is handled by API as well.

The system has another server that is *human resource management server*. This server provides the required resources for human computation. The resource allocation and usage using this component are handled by HTTP API calls too. The application server does the API calls.

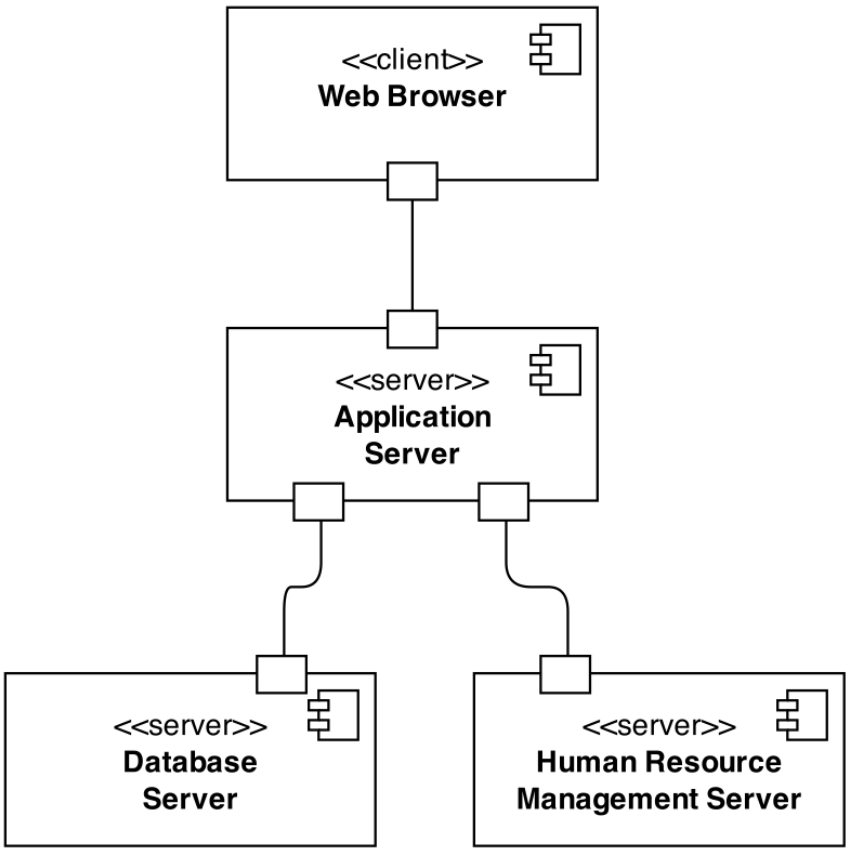

Figure 3.3: Architecture of the platform.

The client-side of *Crowdy* is where crowdsourcing applications are developed. Client part consists of two components: *workflow editor* and *static analyzer*. The *workflow editor* sits on top of *static analyzer* as shown in Figure 3.4 and uses it's services to analyze applications.

The *workflow editor* is where applications are designed. The editor provides users a list of available operators and a flow composition panel. An application is a flow where information flows from source to sink operators. An operator can be created by dragging from the list to the flow composition panel. A flow is created by connecting one operator to another.

The *static analyzer* runs all the time during the application development. This component checks the validity of the application. Warnings and errors are raised when user tries an action that can result in an invalid application. When user is

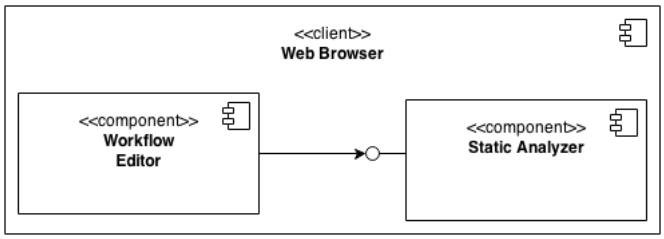

Figure 3.4: Client-side architecture.

ready to submit application to the server-side for execution, *static analyzer* does a final validation check. If validation succeeds, application can be submitted to the server-side and execution can be started.

The server-side of *Crowdy* is where crowdsourcing applications are executed and results are generated. Server part consists of three components: *API*, *virtual machine* and *service adapters*. In addition, server-side is connected a database server and a crowdsourcing platform server. Figure 3.5 demonstrates the logical decomposition of these components and their usage relation.

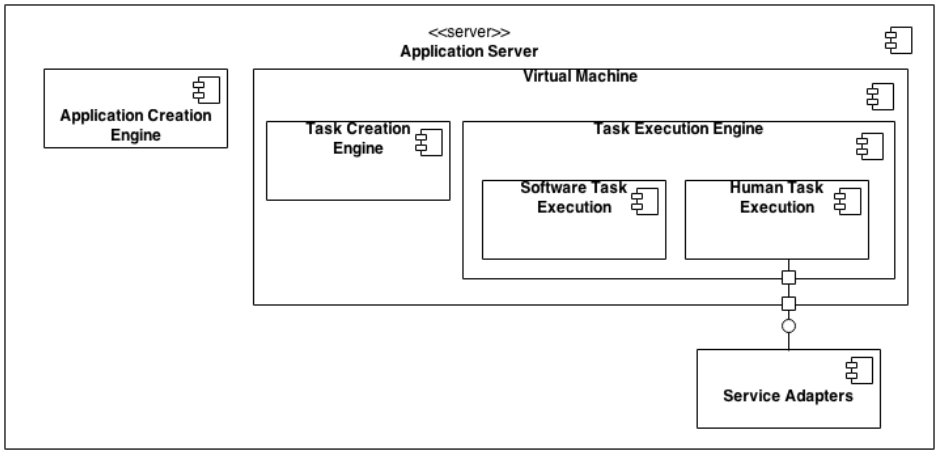

Figure 3.5: Server-side architecture.

The *application programming interface (API)* on the application server specifies how client-side should interact with server-side. Client-side submits application and receives the details of an application state via *API* using remote HTTP calls. *API* creates and updates applications. Application states are saved into *database server*.

The *virtual machine* is a component responsible for executing applications. It provides service-independent environment for human computations. The *database server* is periodically checked, processes are created, and required computation resources are allocated by *virtual machine*. These processes are then executed and results are saved again into *database server*. *Virtual machine* uses the server itself to execute processes that needs software resources. The *service adapters* component is utilized to allocate human resources.

The *service adapters* component provides an interface to *virtual machine* to execute processes that require human computation. This component uses and adapts *API*s provided by external crowdsourcing platform servers. The requests from *virtual machine* are translated into requests that external APIs can understand and submitted there. After process completion, results are gathered and saved into *database server*.

The *database server* is where application state is kept. It is accessed by *API* and *virtual machine* to create new applications and update application state correspondingly.

Figure 3.6 demonstrates the architecture in detail via components and connectors. As explained before this is an client-server architecture in which the connections in between web services are handled by APIs over HTTP. Considering architecture from this perspective reveals the significance of *virtual machine* within *application server*. *Virtual machine* connected to *database server* and *service adapters* constitutes the idea of stream-based human computation.

Applications created by *application creation engine* via the submissions from *workflow editor* are saved into *database server*. As mentioned before *virtual machine* periodically checks *database server*, and either creates new tasks or executes the existing ones. In that sense, *virtual machine* has two basic components: *task creation engine* and *task execution engine*. *Task creation engine* retrieves newly submitted applications and creates the initial task, which conforms to jobs of *source operator* (see concept explanations later in the chapter). Task created by *task creation engine* is saved back to the database with the information required for task execution. *Task execution engine* receives the available tasks

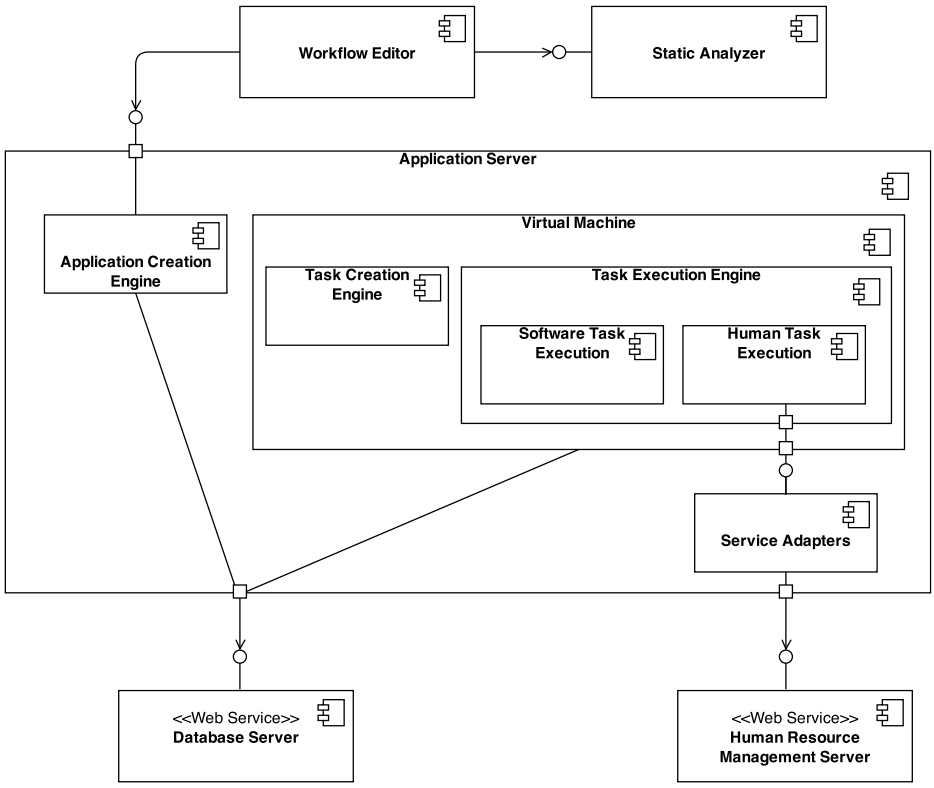

Figure 3.6: Detailed architecture in terms of components and connectors.

from *database server*. The task is tagged as software-related or human-related based on the details. If task is software-related such as saving a text into a file or sending an email, then it is executed by *software task execution* component. Otherwise, task is human-related and that means this specific task requires human resources. Therefore, task is received by *human task execution* component. This component decides the necessary resources for execution and allocates them via *service adapters* component, which accesses *human resource management server* via it's API. Task is then executed by human workers and results are sent back from *human resource management server* to *human task execution* component via *service adapters*.

3.1.1.3 Allocation Views

As it is mentioned earlier in the section *Crowdy* platform is a REST architecture. Figure 3.7 demonstrates how the platform is deployed and its relations with

nonsoftware elements.

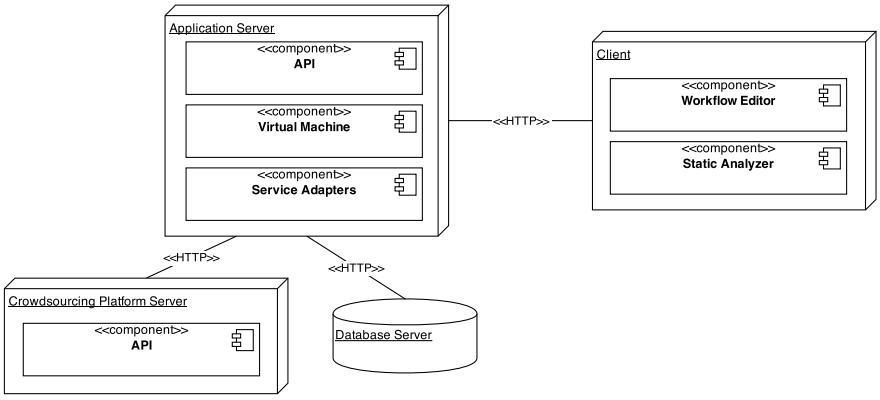

Figure 3.7: Architecture of the platform.

In the remainder, the fundamental concepts of *Crowdy* are explained in more detail and features are explored as we look into various aspects of application development using *Crowdy* platform. In Chapter 4 the tool developed over these concepts is explained, and a sample application is developed using the tool.

3.2 Application Development on the Platform

Crowdy applications are developed to solve complex and sophisticated problems that require both human intelligence and computing power. A typical application contains three main high-level components: data ingest, processing and data egress. Indeed, the pattern of interaction that characterized by successive transformations of data streams aligns with pipe-and-filter style [33].

Data arrives at operators at input ports, is processed and then passed to the next operator in the downstream via its output ports though flow. The operators in this work corresponds to the filter definition in pipe-and-filter style, and the flow is consistent with pipe definition [33] as demonstrated in Figure 3.8. The corresponding operator and flow concepts in Crowdy context are described in the following.

Figure 3.9 shows the metamodel for which a *Crowdy* application is based on.

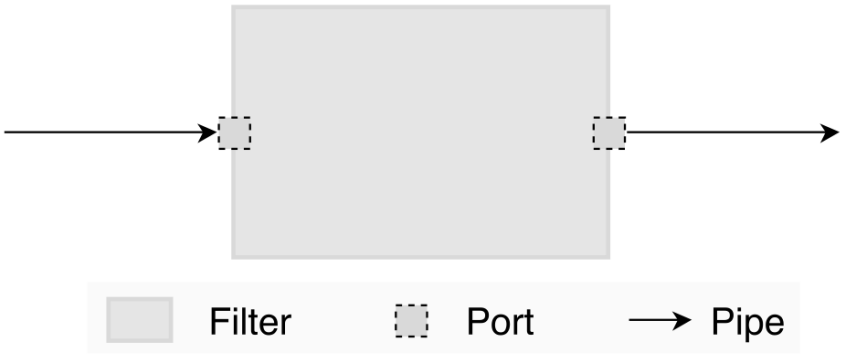

Figure 3.8: Crowdy application correspondence to Pipe-and-Filter style.

An application is formed by a set of *operator*s. As mentioned before, typically an application has three *operator*s: one for inputting data, one for processing that data items, and finally one for outputting the results. However, it is possible to have an application with only two *operator*s: one for data input and other for data output.

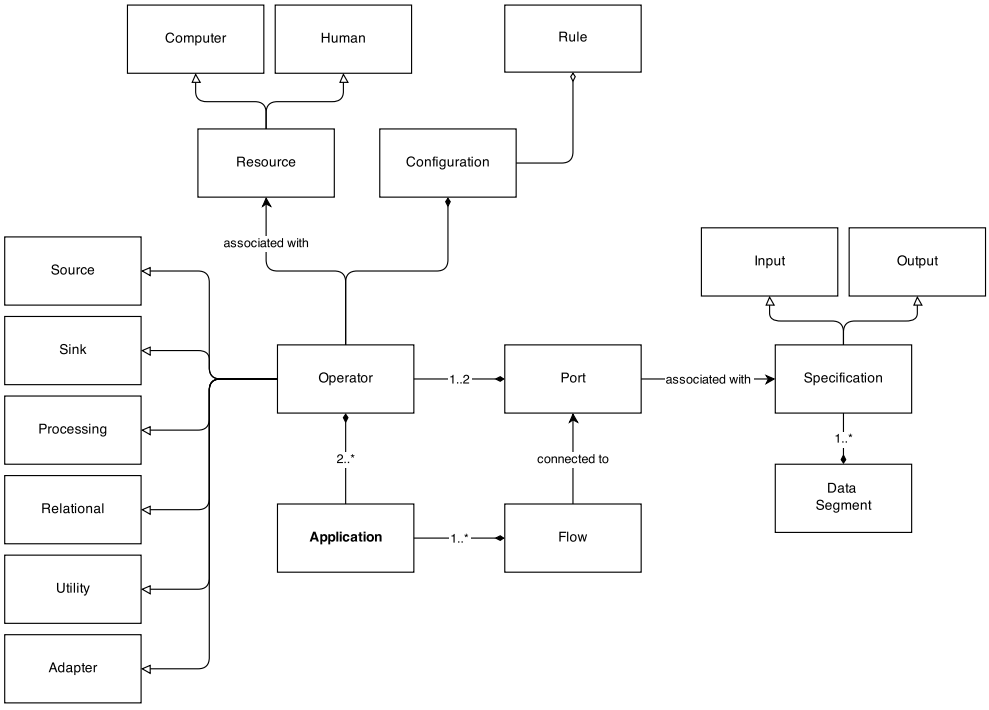

Figure 3.9: Metamodel for a Crowdy application.

Each *operator* is associated with a type that can be one of *source, sink, processing, relational, utility, adapter*. Based on it's type, operator may have one or

two *port*s. The number of *port*s is basically decided by the type. For instance, *source* and *sink operator*s have only one *port*, but others have two *port*s. A *port* is associated with a *specification*, which can be either *input* or *output specification* again based on it's type. *Specification* is formed by *data segment*s that are used to convey information through *flow*s. This means *flow*s are the components that deliver information from one *operator* to another via *port*s.

Operator's type also determines the type of resource it is associated with. The base promise of this study is to combine limited computer and human resources together to solve complex problems. Each *operator* is linked to a *resource*, which can be either *computer*s or *human*s. While that *operator* is executed, related *resource*s are allocated. An *operator* also has *configuration* dependent on it's type. *Configuration* may have *rule*s that decide how *operator* will function based on the data.

Figure 3.10: A sample, minimal *Crowdy* application.

Let's consider a minimal "Hello World" application that has these three components in total, connected in a simple pipeline topology. Figure 3.10 demonstrates the application in the form of a flow graph. On the ingest side, there is a *source operator*, which acts like a data generator. Source operator produces data tuples that are processed down the flow by the *processing operator*. Finally, the *sink operator* simply converts the tuples in such a form (text file, email etc) that can be easily interpreted by requesters.

The application flow graph is specified as a series of operator instances and connections (data flows) that link them. A data flow basically transfers data tuples produced by an operator to another. One or more data segments can be assembled in a data tuple via output specification of an operator instance (see Section 3.3). In addition, several options can be specified to configure an operator instance. These include parameters, operator-specific rules, which are studied in the rest of this chapter.

In a more realistic application, one or more source operators can be employed to produce various data tuples that differ in both size and specification. Similarly, the application would have one or more processing operators along with other types of operators organized in a way that is significantly more complex than this example.

3.2.1 Operator

An operator is the basic building block of an application. Operator has a type that is specified at the time of creation. This type determines configuration respectively. Also a unique ID is assigned to an operator. In addition, operator has optional name and description fields that can be used for bookkeeping purposes.

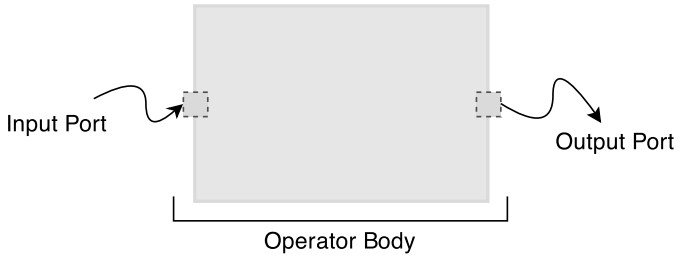

Figure 3.11: Base operator representation.

Operator may have an input port or output port or both corresponding to it's type definition. Figure 3.11 demonstrates a base operator, which consists of a body and ports. Although it is not shown here, each operator presentation has a specific icon on their body associated with it's type.

An operator may output tuples to any number of operators, but it can only receive tuples from one operator unless it is a union operator (see Section 3.2.1.5). Union operator can receive tuples from multiple operators and aggregate them, if these tuples have the same specification. Therefore, consistency of incoming flow specification for each operator type is ensured. This is a significant feature to guarantee operators functionality, because an operator (excluding source operators) uses and operates specifically on the information from incoming flow.

Crowdy provides a set of built-in operators that can be used to build applications. In general, these operators perform common tasks associated with data generation, processing and outputting.

Operators are generally cross-domain to allow general-purpose computation possible. They are grouped under six main categories: *source, sink, processing, relational, utility, adapter.*

3.2.1.1 Source operators

The set of source operators generates data tuples. These operators do not have an input port, but have an output port, which produces data tuples. Figure 3.12 represents a source operator.

Figure 3.12: Source operator representation.

Source operators together with processing operators are the ones that can be used to specify data flow coming out of an operator. Output specification is an action to identify the data tuple with a series of segments. Other operator types cannot make changes on output specification, but can manipulate the flow by dropping or copying data tuples.

human. The *human source operator* is a stateless operator used to produce new data tuples via human workers. Existing crowdsourcing services such as MTurk is used to produce new tuples. A new data tuple is produced per successfully completed human intelligent task. These tasks are automatically created and posted with respect to the specified parameters of the operator.

Human source operator has the following parameters:

- `number of copies`: The maximum number of data tuples can be generated

by the operator. It's value ranges from 1 to 1000.

- `max allotted time`: The maximum time in seconds given to a human worker to solve and submit the task. It's value ranges from 10 to 300.
- `lifetime`: The life time of human task in hours during which period task will be available to human workers. It's value ranges from 1 to 72.
- `payment`: The payment in cents to be given to the human worker in case of successful task completion. It's value ranges from 5 to 500.
- `instructions`: The detailed information for human workers on how to complete the task.
- `question`: The set of sentences asking for specific information from human workers.
- `input list`: The list of `input`s that will be shown to human worker to fill in. An `input` can be a type of *text, number, single choice* or *multiple choice*. Each of these types corresponds to an HTML element. Table 3.1 lists types and their details.

  A text-input presents an input field where the human worker can enter data. The maximum number of characters that can be entered to the field can be set by the requester. Similarly number-input corresponds to a input field where only numbers can be fed in, and the maximum and minimum value for the field are set by the requester. The other two input types conform to input fields where the options given by the requester are presented to the human worker as a list. Human worker is expected to select only one and one or more options for single choice and multiple choice types respectively.

  In fact, each `input` conforms to a segment in the data tuple.

manual. The *manual source operator* is a stateless operator to produce new data tuples.

Manual source operator has the following parameters:

Table 3.1: List of inputs and options

type	parameters	HTML element
text	max number of characters	input [type=text]
number	min value, max value	input [text=number]
single choice	options	input [type=radio]
multiple choice	options	input [type=checkbox]

- `manual entry`: The manual text to be parsed and used to produce new tuples.
- `delimiter`: Delimiter to determine segments in a tuple. This can have one of the following values: none (`''`), white space (`' '`), tab (`'\t'`), comma (`','`), column (`':'`).

Manual source operator uses manually entered text to create new tuples. Operator retrieves the manual text, parses it line by line and then applies the delimiter. Therefore, each line constitutes a data tuple, and delimiter is used to create segments in a tuple.

For example, if `delimiter` is chosen to be white space and the following is entered to `manual entry`,

```
Lorem ipsum
Consectetur adipiscing
Phasellus vehicula
```

the following data tuples will be generated:

```
[
{"segment_1": "Lorem", "segment_2": "ipsum"},
{"segment_1": "Consectetur", "segment_2": "adipiscing"},
{"segment_1": "Phasellus", "segment_2": "vehicula"}
]
```

It is possible to have such a manual entry that ends up in different number of segments for different lines. To prevent this happening manual source operator uses the first line to generate output specification. If more segments are generated

in the following lines, they are discarded. If there are not enough segments in another line, then the corresponding segments are emptied and then outputted.

3.2.1.2 Sink operators

The set of sink operators is where data tuples are serialized and converted into the formats that can be used by requesters with ease. These operators have one input port, but no output port. Figure 3.13 demonstrates a sink operator.

Figure 3.13: Sink operator representation.

email. The *email sink operator* is a stateless operator to convert data tuples into a text format and email them to requesters. `email` parameter specifies the requester's email address.

file. The *file sink operator* is a stateless operator to serialize the data tuples into a file. Operator has one parameter `filename` that is used to specify the name of file in which tuples will be written.

3.2.1.3 Processing operators

The set of processing operators provides data tuple processing via human workers. These operators have both input and output ports. Figure 3.14 shows a processing operator.

Figure 3.14: Processing operator representation.

As mentioned before, processing operators can manipulate the data flow specification in addition to source operators. These operators can change the existing flow specification by adding, deleting or editing data segments.

human. The `human processing operator` is almost same as human source operator. The difference is human processing operator has an input port. That means there is a flow of data tuples coming to operator. These incoming tuples are made available to requesters via their specification.

The parameters of human processing operator is no different than the parameters of human source operator. Additionally processing operator has available segment list, which provides placeholders for the segments of an incoming data tuple. Requesters can place these placeholders in `instructions`, `question` and `input list` (applicable to single choice and multiple choice inputs).

At runtime when a new tuple arrives, each placeholder in parameters is replaced with the corresponding value of incoming tuple's segment. This enables dynamically created human tasks. Therefore, requesters can create an information flow from one operator to another.

3.2.1.4 Relational operators

The set of relational operators enables fundamental manipulation operations on the flow of data tuples. Each relational operator implements a specific functionality providing continuous and non-blocking processing on tuples. Therefore, these operators have both input and output port.

selection. The *selection operator* is a stateless operator used to filter tuples. A typical selection operator is shown in Figure 3.15.

Figure 3.15: Selection operator representation.

On a per-tuple basis a boolean predicate is evaluated and a decision is made as to whether to filter the corresponding tuple or not. Boolean predicates are specified by requesters as part of operator parameterization. These predicates, which are identified in `rules`, are the only members of parameters.

A selection operator has zero or more rules to filter data tuples. When there is no rule specified, then no filtering will be done, and all data tuple will be passed down to data flow. Otherwise, each rule is evaluated on an incoming data tuple. Whenever a rule is evaluated to be true, then corresponding action is carried out that is either filter in or out the tuple, and the rest of rules is discarded. If no rule is evaluated to be true on a data tuple, then tuple is still passed to the next operator(s). Rules share the following predefined format:

Filter (in/out) when `boolean-predicate`

Similarly `boolean-predicate` has the following format

`segment-name` *(equals/not equals/contains)* `query`

where `segment-name` is one of the segments of incoming tuple, and `query` is a free-text to be filled by requester.

sort. The *sort operator* is a stateful and windowed operator used to first group tuples and then sort them based on the specified data segment and order. Figure 3.16 illustrates the operator.

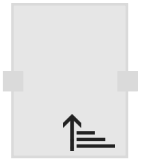

Figure 3.16: Sort operator representation.

Sorting is performed and results are produced (outputted one by one) every time window trigger policy fires. The policy is basically triggered when the number of data tuples reaches `window size`, which is specified as a parameter. `window size` can have a value between 1 and 100.

Similar to selection operator, sort operator implements a set of rules that simply identifies the segment and the order to be used for sorting. If no rule is specified, then tuples are sorted in the ascending order with respect to the first segment in the incoming data tuples. If there are more than one rule is given, then these rules are applied in the order they are specified by requester.

Sorting rules share the following predefined format:

Sort using `segment-name` *in (ascending/descending) order*

where `segment-name` is one of the segments of incoming tuple.

3.2.1.5 Utility operators

The set of utility operators provides flow management functions. These operators handle operations such as separating a flow into multiple flows or joining multiple flows into a single one.

enrich. The *enrich operator* is a stateless operator used to enrich data flow by replaying incoming data tuples. Figure 3.17 shows an example view of the operator.

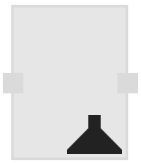

Figure 3.17: Enrich operator representation.

Enrich operator has one parameter called `number of copies`. This parameter determines how many copies will be produced for each incoming tuple. It's value ranges from 1 to 10.

split. The *split operator* is a stateless operator used to divide an inbound flow into multiple flows. This operator has one incoming flow and and can have one or more outgoing flow. Figure 3.18 provides the presentation of a split operator.

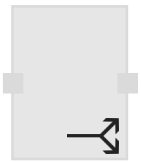

Figure 3.18: Split operator representation.

The boolean predicates specified in rules are evaluated whenever a new tuple arrives. Then, a decision is made to where to send the tuple. Similar to relational operators, rules are specified by requesters as part of operator parameterization.

A split operator has zero or more rules. When there is no rule specified, then tuples are passed to every operator connected down the flow. Otherwise, each and every rule is evaluated on an incoming data tuple. Whenever rule's predicate is evaluated to be true, then tuple is passed to the corresponding operator. If no predicate turns out to be true for a tuple, then it is dropped.

It is possible that more than one rule for the same `next-operator` can be true for a specific data tuple. This doesn't mean that tuple will be sent to that operator multiple times. Only one tuple will be outputted to `next-operator`.

Rules share the following predefined format:

Send to `next-operator` *when* `boolean-predicate`

in which `next-operator` is the connected operators down the flow, and `boolean-predicate` has the same definition given for selection operator.

union. The *union operator* is a stateless operator used to join two or more data flows into one. Different than other operators, this operator can receive more than one flow. These flows are combined by the operator and outputted as if they are a single flow. Figure 3.19 demonstrates the operator.

Union operator requires incoming flows to have the same specification. Otherwise, union operation will basically fail.

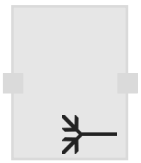

Figure 3.19: Union operator representation.

3.2.2 Flow

Flow is the connection between operators used to move information from one operator to another. A flow is formed by one or more data segments. A segment has an identifier and corresponding value. In that sense, flow is like a hash table where segment identifiers are keys, and values stored in segments are values as demonstrated in the following:

```
{
"segment-1": "value-1",
"segment-2": "value-2",
...
"segment-N": "value-N",
}
```

The details of a flow can be specified by either source or processing operators. It is immutable in other operators. Specification can be achieved via output specification section in operator configuration. Requesters can create new segments, edit or delete existing ones. Source operators are useful to specify flow specification in the early stages of workflow design and generate tuples using that specification. The flow specification is crucial, since it determines the specifics of the information that is carried over in the application. Still this specification can be redefined in processing operators.

3.3 Flow Composition

The flow composition using *Crowdy* can be achieved by simply creating new operators, configuring them and connecting them together. However, there are

certain predefined rules that should be satisfied to develop a valid application flow and execute it.

An application is formed by a workflow. Although it is not suggested, application can have multiple workflows. A workflow is basically a set of operators connected together. An operator can be connected to another operator by attaching a flow from the output port of an operator to the input port of the other operator. Workflow must contain at least one source and one sink operator. There may be operators that are not connected to any other operator within the application. Operators with no connections are discarded and they do not cause validation failures.

A valid workflow means that each each operator employed in that workflow is valid. Validity of an operator depends on it's type and therefore it's configuration. In the following each operator group is examined in terms of validation.

3.3.1 Source operators

A source operator has one output port, which outputs the data tuples with given specification. Therefore, source operator is required to have an output specification that has at least one data segment in it.

human. Parameters of a human operator should be valid. That means `number of copies` is an integer between 1 and 1000, `max allotted time` is an integer between 10 and 300, `payment` is an integer between 5 and 500. Although `instructions` can be left empty by the requester, `question` cannot be empty. There must be at least one `input` defined in the `input list` to satisfy that source operators have an output specification with at least one segment defined.

manual. `manual entry` and `delimiter` should not be empty. Nonempty `manual entry` and `delimiter` correspond to an output specification with at least one segment in it.

3.3.1.1 Sink operators

A sink operator has one input port, which receives data tuples.

email. Operator must have a valid email address given in `email` parameter.

file. Operator must have a valid file name given in `filename` parameter.

3.3.2 Processing operators

A processing operator has one input and one output port. It receives data tuples, processes them and outputs them to next operator down to flow. Similar to source operators, processing operators must have an output specification that has at least one data segment in it. Since human operator is the only processing operator, the validity rules for this operator is same as human operator of type source operator.

3.3.3 Relational operators

A relational operator has one input and one output port. It is useful to manipulate flow. Operators of this type has rules to determine manipulation.

selection. Each rule in the rule list must be valid, which means a rule with valid `segment-name`.

sort. Operator must have a valid `window size`. This value can be an integer between 1 and 100. Additionally each rule in the rule list must be valid by being formed by a valid `segment-name`.

3.3.4 Utility operators

A relational operator has one input and one output port. This type of operators is useful for flow management.

enrich. Operator must have a valid `number of copies`. This value can be an integer between 1 and 10.

split. Split operator must have an incoming flow to-be-splitted and at least one outgoing flow. Split operation is defined by rules. Each rule in the rule list must be valid by specifying a valid `next-operator`.

union. Union operator must have at least one incoming flow to-be-merged and one outgoing flow. Each incoming flow must have the same specification, which is ensured by application editor by preventing connecting flows with different specification to the same union operator.

Chapter 4

Tool

Crowdy has been implemented as a web-based tool and made freely available[1]. The client part of the tool is developed using Javascript, while application server and virtual machine is written in python. The tool is deployed to Heroku [2] cloud application platform. The tool can be used by anyone to solve a problem that requires both software and human resources. In terms of crowdsourcing parlance users of the tool is called *requesters*.

Requesters can use *Crowdy* to define and configure crowdsourcing applications. In this section, the application development over *Crowdy* platform is demonstrated using the tool. The minimal example application presented earlier in this section is implemented. The steps are presented and explained in detail with the sample screenshots from the tool.

A *Crowdy* application consists of operators in which one is connected to the other creating an information flow. Figure 4.1 shows the flow composition panel of the tool, which is used to create and configure applications. At the top (pointed by (1)), a simple set of instructions is listed to help users in the application development process. Additionally links to validate an application, clear the flow composition panel and report a bug are given here. On the left (pointed by

[1]http://crowdy.herokuapp.com
[2]http://www.herokuapp.com

(2)), the list of operators is given and that list is separated by operator type, which is described in detail earlier in this section. The panel (pointed by (3)) is where operators are placed and linked together to develop an application. Finally, the messages panel (pointed by (4)) is where important information about the application are displayed to user such as warnings, validation failures or successes etc.

Figure 4.1: Flow composition window.

The creation of an operator is simple in *Crowdy*. Operators are created by dragging and dropping the specific operator from the list to the panel. An operator can be moved across the panel freely by dragging. An operator can be selected and deselected by clicking the operator body. Once an operator is selected, the border color of the operator is darkened to indicate it is actually selected. A selected operator can be removed by pressing backspace key in the keyboard. Figure 4.2 shows that a source operator is created and selected.

As explained in Section 3.2.1 in a detailed manner, operators in *Crowdy* are easily configurable. Each operator has it's own definition of configuration and list of options. This configuration window can be opened by double-clicking operator body. Figure 4.3 demonstrates the configuration window for *source manual operator*. The configuration has three sections: *details*, *parameters* and

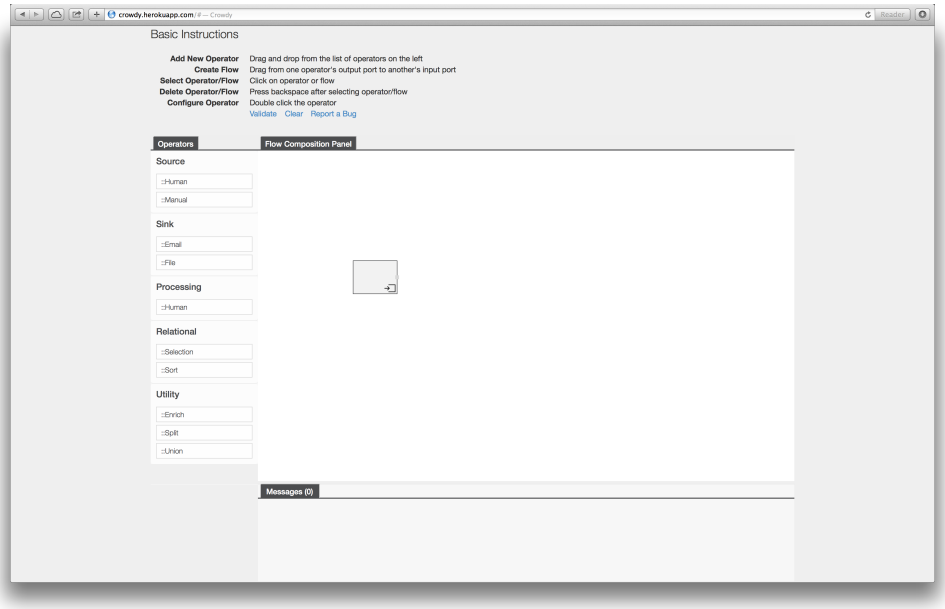

Figure 4.2: An source operator added to flow.

output specification. The *details* section exists for every operator and it can be used for bookkeeping purposes such as assigning names, noting some information specific to those operators. The *parameters* section is specific to each operator and this is where actual configuration per operator can be achieved. In this example, configuration for *source manual operator* is depicted. Finally, the *output specification* section is where output definition of that operator is shown. This section lists the data segments for that operator. These segments can be renamed by clicking and editing the names. This section is not shown for *sink operators*, since they don't have any output to be delivered to some other operator. In fact, this section is automatically generated for *relational* and *utility operators* and updated once the source of those segments makes any changes, since they are able capable of changing the output specification by adding or deleting new segments.

Figure 4.4 redemonstrates the configuration window, but now fields are complete and outputs are specified. To exemplify the functionality the list of countries and capitals separated by tab is entered. Delimiter is selected to be tab in that sense and output specification is automatically generated. The data segments are renamed after the information they are containing. The segment names will be

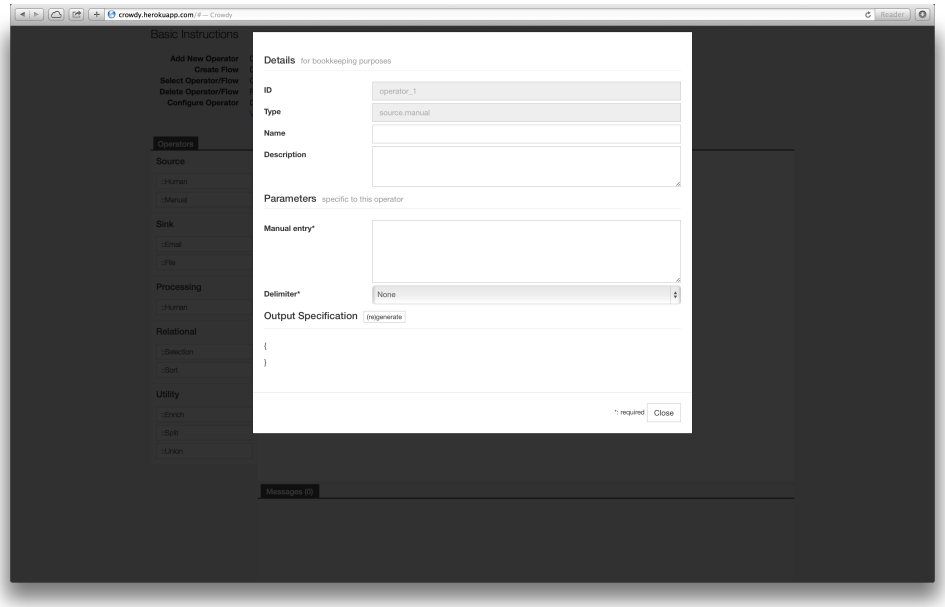

Figure 4.3: Configuration window for source manual operator.

useful down the flow while creating human tasks or manipulating flow of information.

The *human processing operator* is created and then the output of source operator is connected to it to create the flow as shown in Figure 4.5. The arrow on the flow shows the direction of information. Similar operator selection and deletion operations, the flow (the link between operators) can be selected and deselected by clicking. A selected flow can be removed by pressing backspace key in the keyboard. Once a flow is deleted, the connected operators to that flow are deleted as well.

Figure 4.6 shows the configuration window for *human processing operator*, which is useful to create human tasks. Comparing this configuration with the previous one indicate the difference in the *parameters* section. This section now presents the configuration specific to this operator. The fields are completed for the sake of demonstration and output specification is created. For instance, the data segment of the previous operator is available here to be able to create customizable questions under *available segments* part. Those segments can be dragged and dropped to *instructions*, *question* and *single/multiple choice* answer

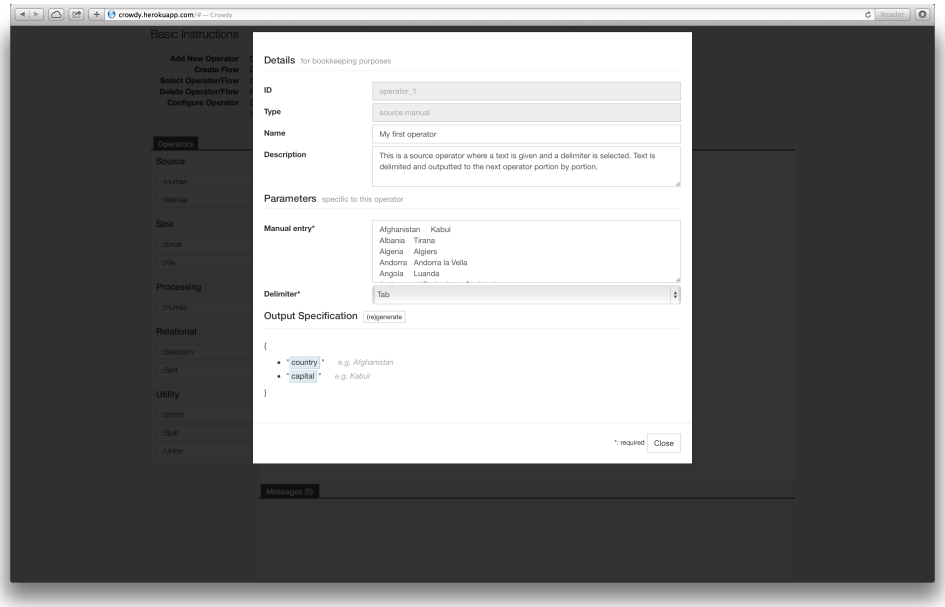

Figure 4.4: Configuration window for source manual operator filled with sample information.

fields. Dropped segments can be easily deleted by pressing X button on the right.

In addition, the human task that is configured in this operator can be previewed by clicking *preview human task* button right near the *parameters* section title. Figure 4.7 shows the corresponding human task for that operator. The segments dropped in question field is presented using a different color. These will be automatically replaced with the segment value once information starts flowing from source to sink, which is when the application is executed.

Finally, a *sink operator* is added and connected to *human processing operator* as shown in Figure 4.8.

The configuration window (demonstrated in Figure 4.9) is intentionally left blank to show validation capabilities of *Crowdy*. Then, validation is initiated by clicking *validate* link at the top of the flow composition window.

The validation result is shown in Figure 4.10. The result lists all warnings and errors. An explanation per warning/error is given as well. Users are expected to resolve errors at least to complete validation and finally submit application for

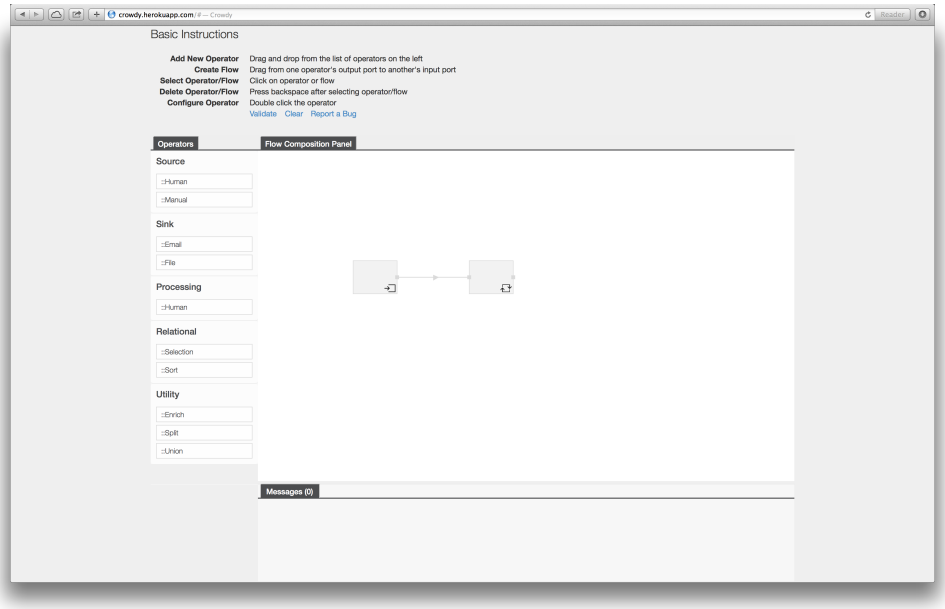

Figure 4.5: Two operators connected via flow.

execution.

When we open the configuration window for *sink operator*, which is the source of validation problems, the problematic fields are marked with yellow for warning and red for error (see Figure 4.11). Since *file name* is required and it is left empty, it caused an error during validation.

At least the errors should be resolved for validation to succeed. Once validation goes through application can be submitted for execution via the link appearing on right bottom of validation window demonstrated in Figure 4.12. After application is submitted, the processes are created and appropriate resources are allocated to solve the problem and find the result. In this specific example, results are expected to be written into a file.

Finally, Figure 4.13 shows the window to report a bug. Bugs can be reported using this window. While custom messages can be given, the tool itself gather particular logs for that session to create an application.

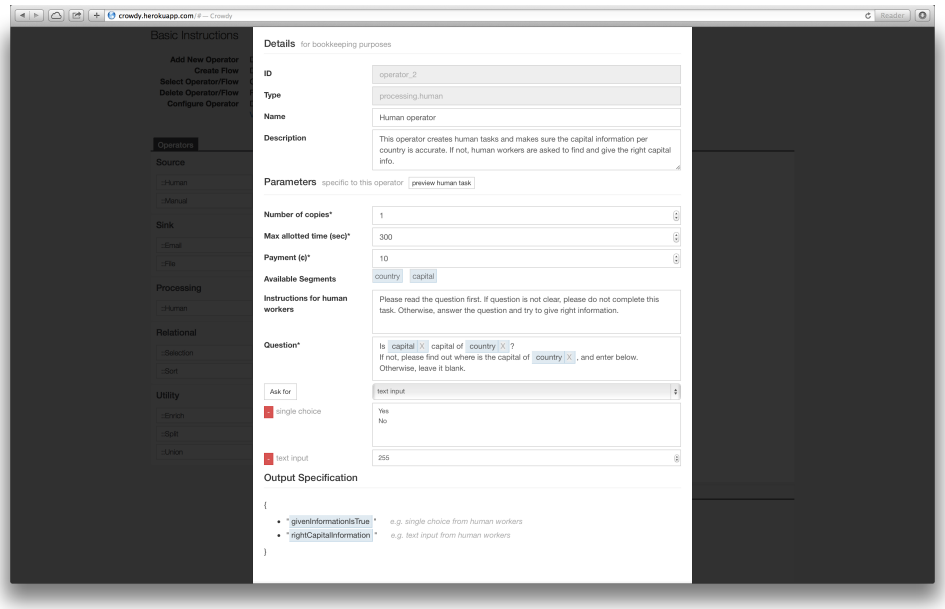

Figure 4.6: Configuration window for human processing operator filled with sample information.

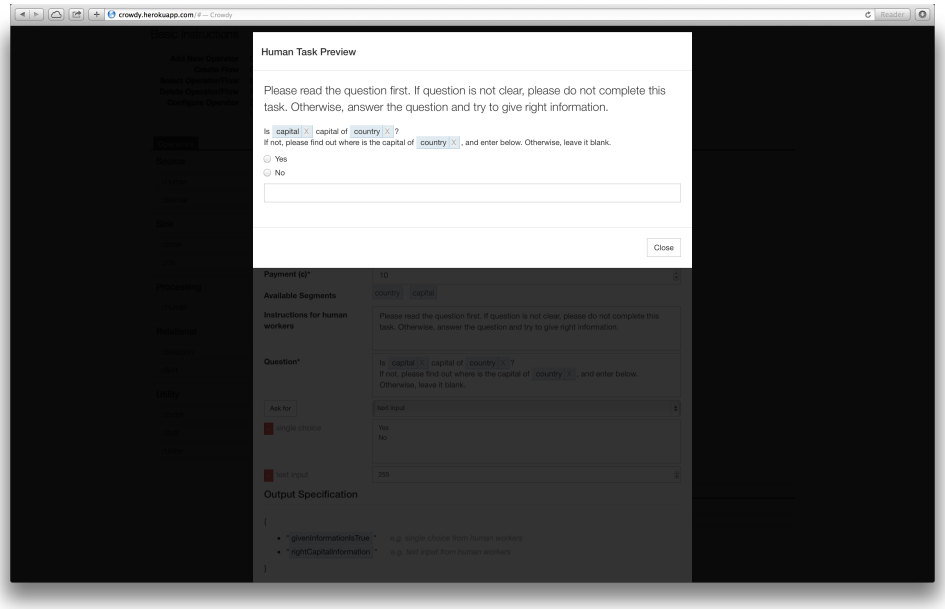

Figure 4.7: Human task preview window for human processing operator.

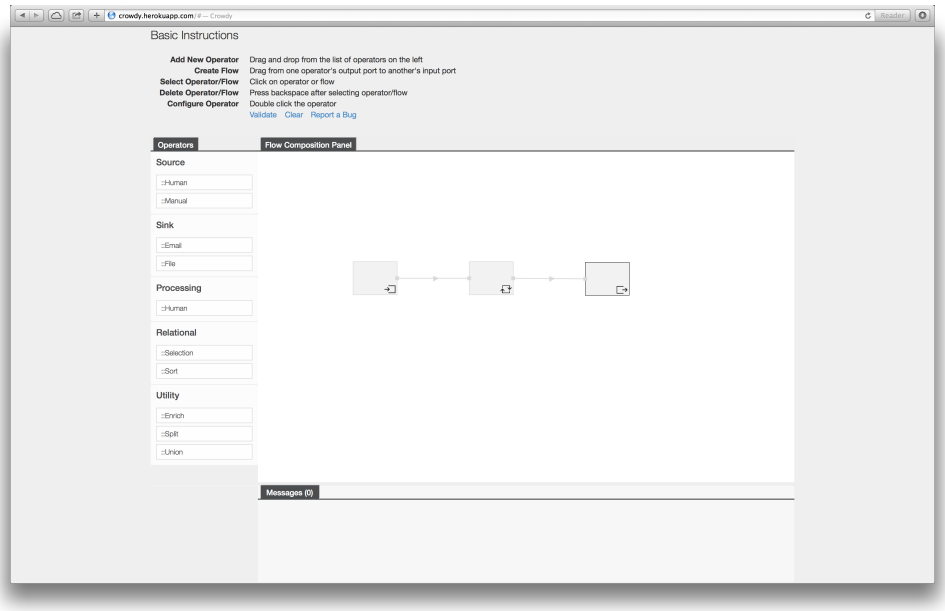

Figure 4.8: A minimal application with source, processing and sink operators.

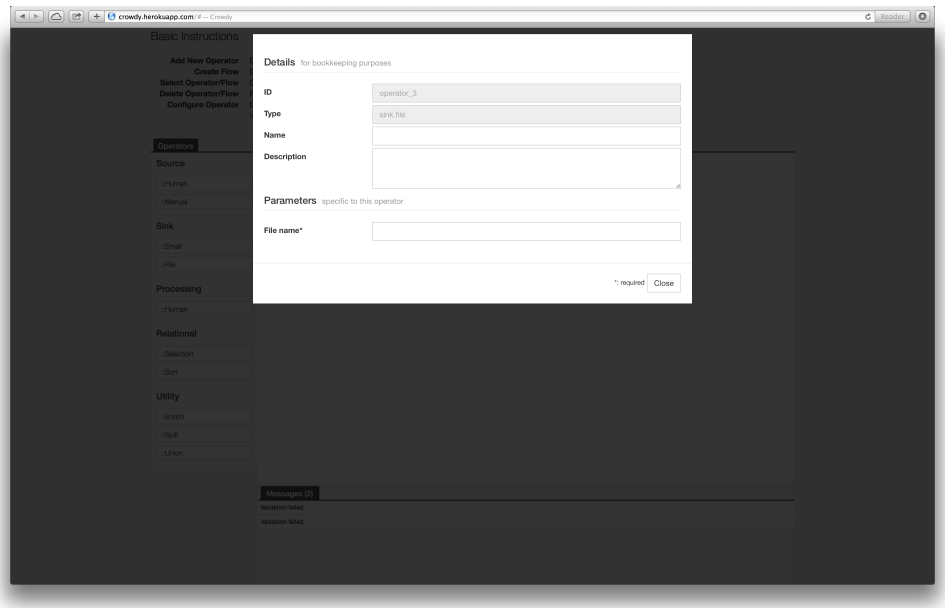

Figure 4.9: Configuration window for sink operator.

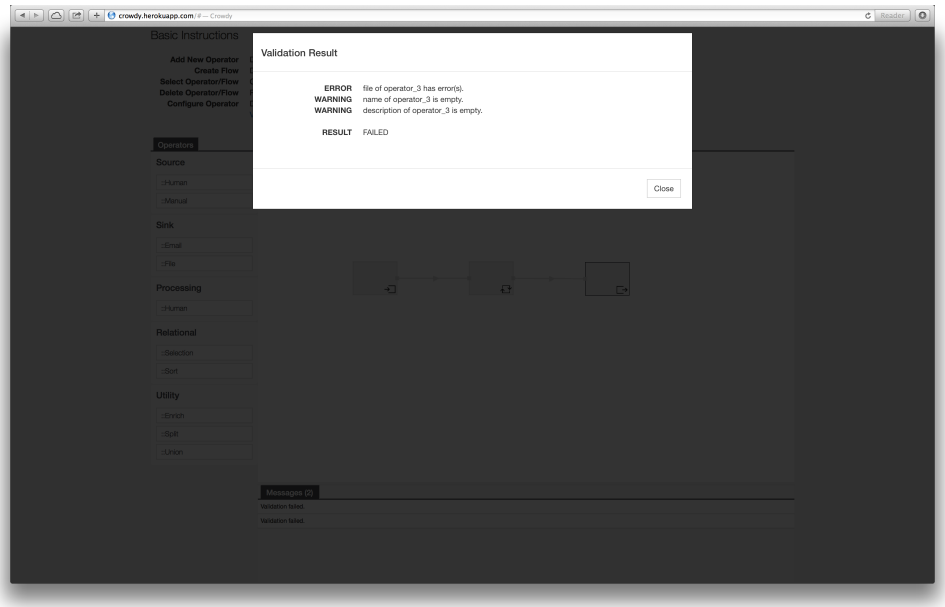

Figure 4.10: Validation windows listing warnings and errors.

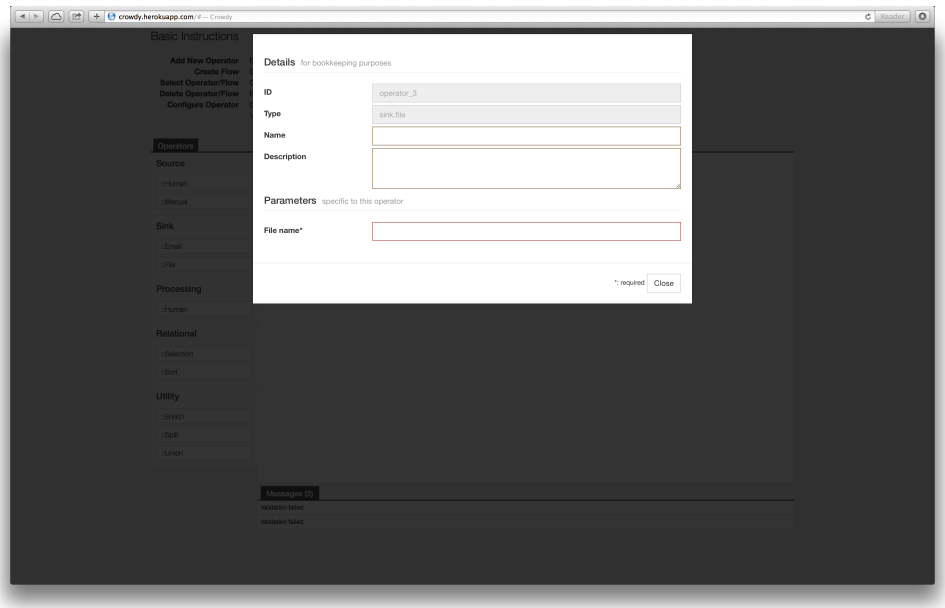

Figure 4.11: Configuration window for sink operator after validation.

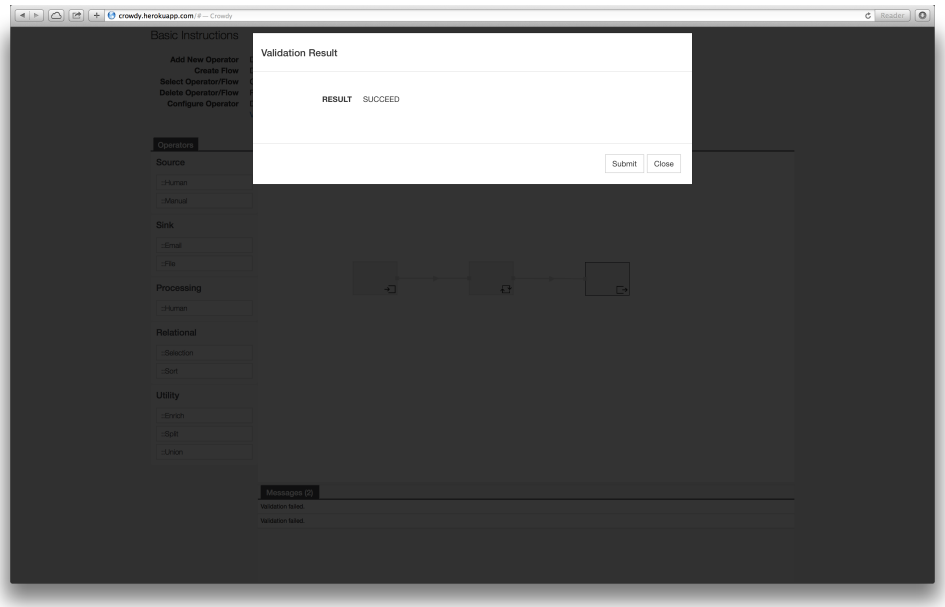

Figure 4.12: Successful validation window with submit link.

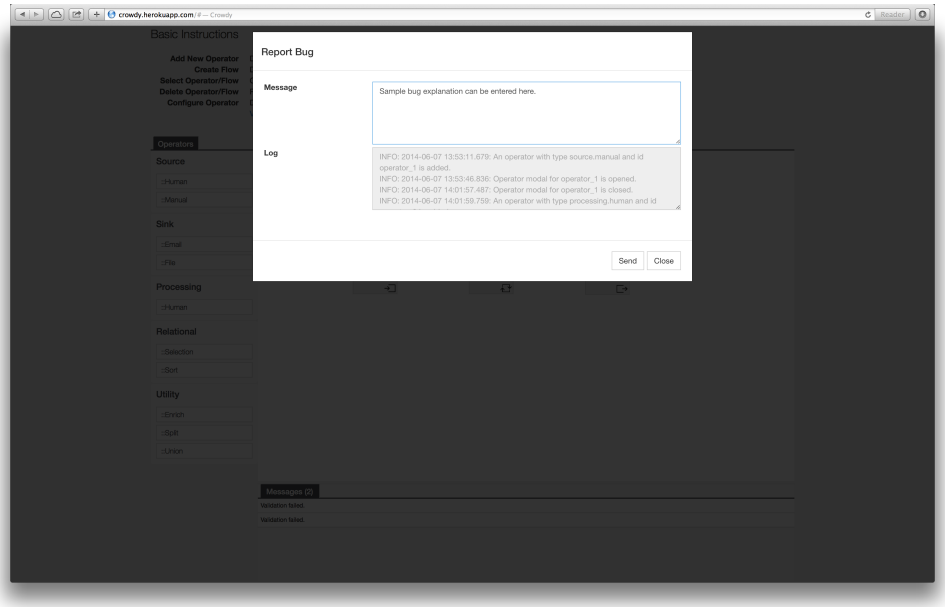

Figure 4.13: Window to report a bug.

Chapter 5

Case Studies

In the following sections, case studies are conducted to provide examples and evidence of how *Crowdy* works in practice. The case studies are picked from two different categories for which crowdsourcing is popularly applied: data verification and translation.

In terms of data verification, crowdsourcing is highly leveraged to help businesses to clean their data and verify it's correctness. In that sense, the first case study considers a company list for which workers are asked to check and correct their information. Translation is another area where crowdsourcing is commonly utilized and the second case study works on a translation problem.

5.1 Verifying Business Information

Finding a company's address or correcting an address is a common scenario that crowdsourcing is popularly applied. In the following, two different cases for this scenario are exemplified and implemented.

Let's assume that we have a list of companies. The list contains company names and their corresponding mailing and website addresses. However, the mailing addresses change when company moves. It is also quite possible that

street names, building numbers can change in time. In fact, web addresses can change too. What we want to do is employ a group of people to check company websites and extract address information and update their addresses if it is different than the ones we have in the list.

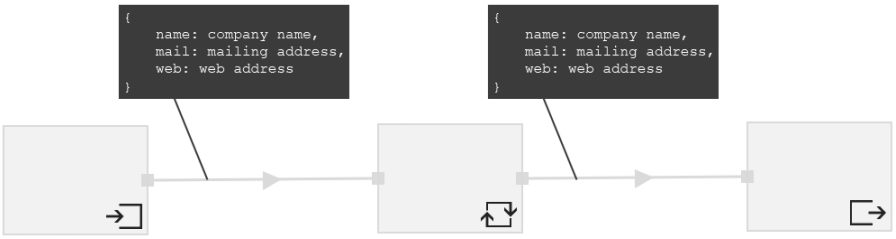

Figure 5.1: Crowdy application to correct business addresses.

The company list is input to the application and processed by human workers. Finally, results are saved into a file. This is presented in Figure 5.1 and detailed in the following operator by operator:

source manual operator. Source manual operator is supplied with company list and the delimiter is selected to be comma. A small part of the list is displayed in Figure 5.2. This operator outputs the data tuples in which there are company, website and mail segments.

```
Company A, 490 E Main Street Norwich CT 06360, www.companya.com
Company B, 70 Cliff Avenue New London CT 06320, www.companyb.net
Company C, 50 Water Street Mystic CT 06355, www.companyc.co
Company D, 15 Cliff Street Griswold CT 06351, www.companyd.com
Company E, 233 River Road New London CT 06320, www.companye.org
```

Figure 5.2: Sample list of companies and their information.

human processing operator. Human processing operator gets data tuples from the source operator. Tuples are used to create the question and ask human workers to check website and mailing addresses. Workers are allowed to work on the task at most 5 minutes and they are given $0.10 per successful completion.

Figure 5.3 displays the question that is shown to human workers.

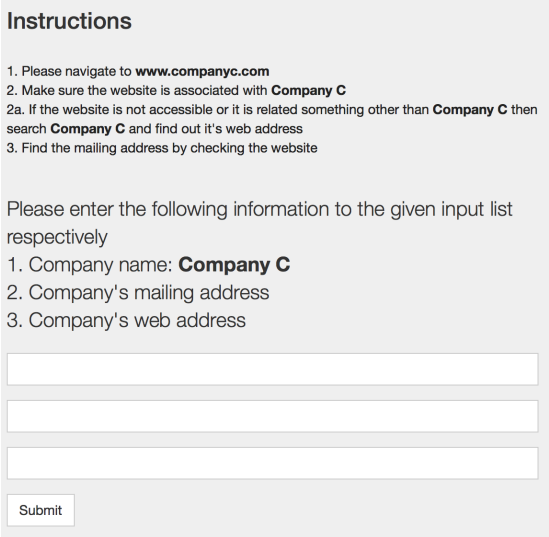

Figure 5.3: Human task that is generated for human workers.

sink operator. Sink file operator receives updated information per company and saves them into a file.

Different than the previous solution, we can follow a different approach to solve this problem. This new approach creates more robust result than the previous approach.

In the human processing operator, human workers are asked to find out and fill mailing and web addresses of companies (see Figure5.4). We can ask two more piece of information from them: whether mailing address if updated and whether web address is updated. The results can be split into four clusters based on the conditions proposed in this approach. The companies that have either updated mailing or web address can be even emailed rather than saving into a file. Figure 5.5 demonstrates the new approach.

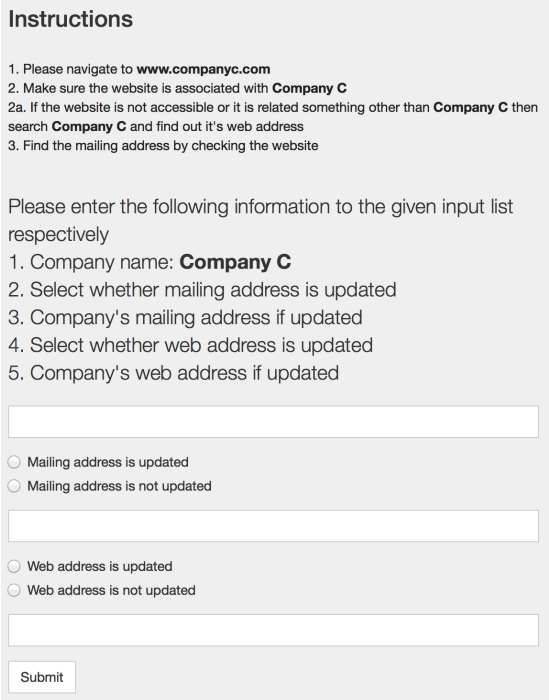

Figure 5.4: Human task that is generated for human workers (updated).

5.2 Translation

Translation is one of the scenarios that crowdsourcing platforms are being challenged, because it is complex, challenging, time-consuming and highly subjective. Translation problem cannot be easily solved by typical human tasks, because tasks are interdependent and parallel approach would not work well on such a scenario.

The concrete problem that we are trying to solve is to translate a Turkish poem to English using Crowdy platform. The input to the application is the famous poet Rumi's poem "Etme" (shown in Figure 5.6) and we expect to get a translated version of it as an output. In the following, a Crowdy application is created to translate this poem into English. The application is improved progressively over a couple of iterations.

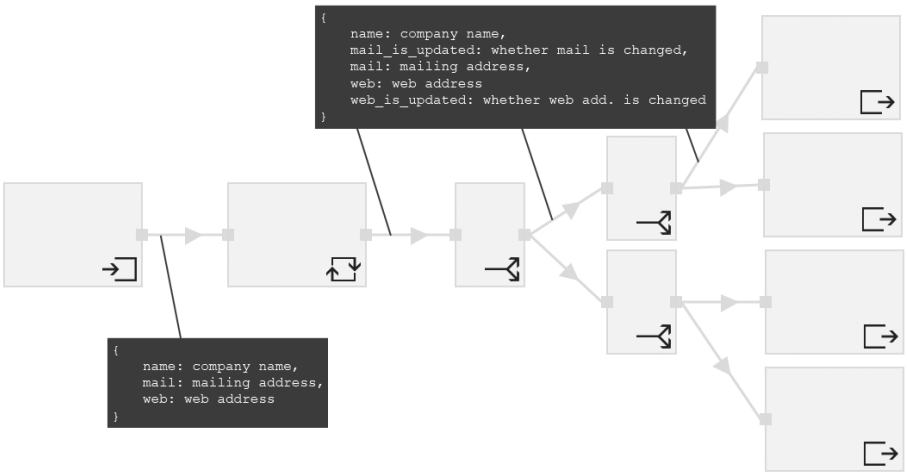

Figure 5.5: Another approach to correct business addresses.

Duydum ki bizi bırakmaya azmediyorsun, etme.
Başka bir yar, başka bir dosta meylediyorsun, etme.
Sen yadeller dünyasında ne arıyorsun yabancı?
Hangi hasta gönüllüyü kastediyorsun, etme.
Çalma bizi, bizden bizi, gitme o ellere doğru.
Çalınmış başkalarına nazar ediyorsun, etme.
Ey ay, felek harab olmuş, altüst olmuş senin için...
Bizi öyle harab, öyle altüst ediyorsun, etme.
Ey, makamı var ve yokun üzerinde olan kişi,
Sen varlık sahasını öyle terk ediyorsun, etme.

Figure 5.6: Some part of Rumi's Poem "Etme".

5.2.1 Naive Approach

The naive approach to solve this problem would be inputing the poem into the application line by line, and asking people to translate a line, and finally saving results into a file. This approach is demonstrated in Figure 5.7 and detailed in the following operator by operator:

source manual operator. Source manual operator is supplied with the poem and the delimiter is selected to be none. Some portion of the poem is displayed in Figure 5.6. This operator outputs the data tuples in which there is one segment called *line* wherein a line from the poem is extracted.

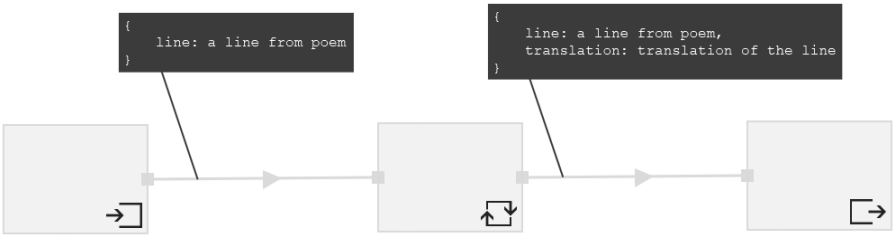

Figure 5.7: Crowdy application to translate a text.

human processing operator. Human processing operator gets data tuples (lines from the poem) from the source operator. These tuples are used to create the question and ask human workers to do the translation. Workers are allowed to work on the task at most 5 minutes and they are given $0.10 per successful completion.

Figure 5.8 presents a sample question that is shown to human workers.

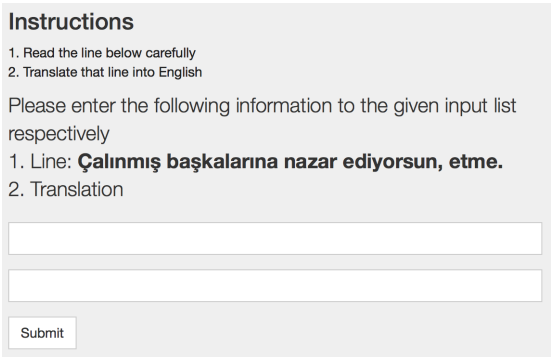

Figure 5.8: Human task that is generated for human workers for translation.

sink operator. Sink file operator receives a line from the poem with it's translated version and saves that into a file.

5.2.1.1 Issues

This approach takes the poem and employ human workers for translation. First of all, there is no quality control. Therefore, low quality assignments are possible and would probably affect the overall result.

Another issue is that the order of tuples received by sink operator is going to be probably different than the actual order. The time that human task is assigned to a worker and the duration that takes for human worker to complete that task cannot be known, although max time allotted to complete the task is set by requester. There is no guarantee that the human tasks are picked up and completed in order. Thus, the file that is created and filled by sink operator the lines from the poem will be in a random order.

5.2.2 A More Sophisticated Approach

The naive approach can be improved by adding an extra segment in the source operator and a utility operator, specifically sort operator, to the application as shown in Figure 5.9. In that way, some of the issues pointed out in the previous iteration can be resolved.

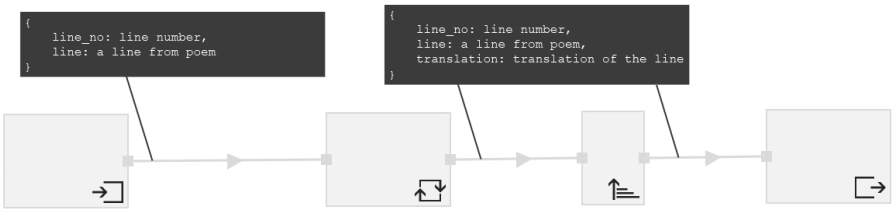

Figure 5.9: Crowdy application to translate a text.

The input in source operator is updated by adding line numbers to the lines. The input corresponding to the one given in the previous iteration (Figure 5.6) is displayed in Figure 5.10. Line numbers are separated by tabs, so now the delimiter is selected to be a tab. Therefore, source operator not only output *line*, but also $line - number$ for the corresponding *line* too.

In addition, sort operator is added to the application in between human operator and sink operator. This operator takes the translated lines from human operator and sorts them with respect to line numbers. The window size of sort operator is set to the number of lines in the poem, so that sorting is done once all the lines are translated. Therefore, the sink operator receives the lines in the order they are given in the poem.

1	Duydum ki bizi bırakmaya azmediyorsun, etme.
2	Başka bir yar, başka bir dosta meylediyorsun, etme.
3	Sen yadeller dünyasında ne arıyorsun yabancı?
4	Hangi hasta gönüllüyü kastediyorsun, etme.
5	Çalma bizi, bizden bizi, gitme o ellere doğru.
6	Çalınmış başkalarına nazar ediyorsun, etme.
7	Ey ay, felek harab olmuş, altüst olmuş senin için...
8	Bizi öyle harab, öyle altüst ediyorsun, etme.
9	Ey, makamı var ve yokun üzerinde olan kişi,
10	Sen varlık sahasını öyle terk ediyorsun, etme.

Figure 5.10: Some part of Rumi's Poem "Etme".

5.2.2.1 Issues

Quality control is still the problem regarding the new approach. The translation done by human workers is not guaranteed to have a good quality. Therefore, quality control is still a fundamental problem.

5.2.3 Final Approach

In this approach, quality control is added to the application. Figure 5.11 demonstrates the application created in the final approach.

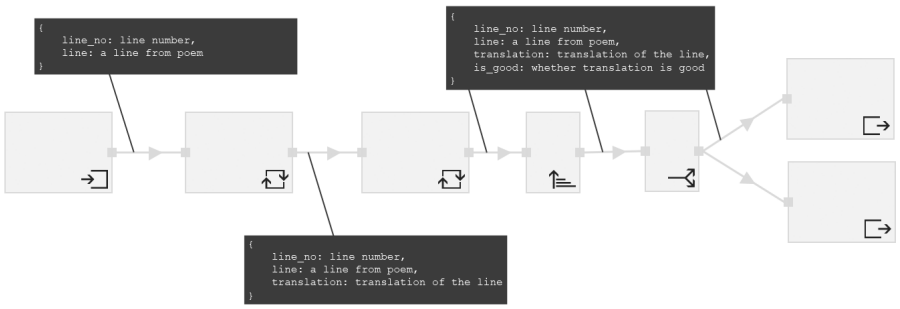

Figure 5.11: Crowdy application to translate a text.

The output from human operator is connected to another human operator that asks people to evaluate the translation done by others. The output from this operator has an extra segment that provides the condition indicating whether

translation seems OK or not. Figure 5.12 shows the question shown to human workers.

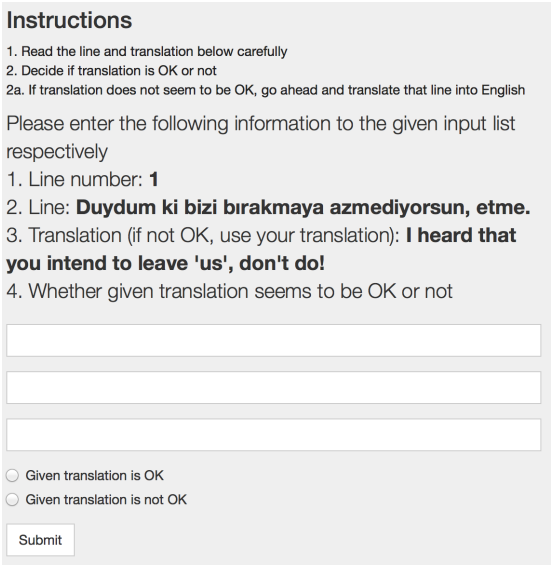

Figure 5.12: Human task that is generated for human workers to check quality of translation.

Additionally a split operator can be placed before sink operator. This operator can take a data tuple and check if it has a good quality translation or not. Using this condition translated lines can be separated into two files. In this way, requesters are able to see which lines are translated well, which are not.

5.2.3.1 Issues

The issues related to quality control are resolved. However, the reliability of this solution can be still questioned, since people are employed to check the quality of the work done by other people. This issue is further discussed in Chapter 7 and possible improvements are listed in Chapter 8.

Chapter 6

Related Work

In response to the challenges of crowdsourcing a number of attempts have been made by the researchers. In the following, the studies related to this work are listed, described and examined in detail. The studies, which try to address the challenges of crowdsourcing and proposing new way of creating crowdsourcing workflows, are considered as related only. Otherwise, the related work list would be too long, since many views Wikipedia and Linux as crowdsourcing systems.

6.1 Crowdsourcing Platforms

The following is a list of crowdsourcing studies that propose a new platform to create crowdsourcing services. Each study is discussed extensively and compared to other studies and *Crowdy* in Table 6.1. The comparison of related works is performed over the building blocks of a crowdsourcing platform that is given in the previous section. The following set of features is chosen to assess related works:

- concept: The concept or paradigm that platform is based on. This basically defines the process of workflow design.

- programming: Requirement of programming for requester to design a workflow.
- user interface: Availability of a user interface to requester to design a workflow.
- human resources: Availability of human resources in task assignment.
- software resources: Availability of software resources in task assignment.
- heterogeneity: Whether heterogeneity of crowd is available to requester or not in task assignments.
- quality control: Existence of a quality control mechanism.

6.1.1 Jabberwocky

Jabberwocky [9] is a social computing stack consists of three main components: Dormouse, ManReduce, and Dog. Dormouse is created to enable cross-platform human and machine resource management. It acts like a "virtual machine" layer in the computing stack, consisting of low-level software libraries that interact with people and computers. ManReduce is a parallel programming framework for human and machine computation working on top of Dormouse. ManReduce is inspired by MapReduce [34] that is basically mapping problem into a set of small chunks of work, and then reducing them on an output that aggregate the responses or solutions. Dog is a high-level programming language on top of ManReduce. The language is formed by a small set of primitives for requesting computational work from people of machines.

Jabberwocky is limited in several ways. The computing stack is a command-line tool supported by restricted built-in libraries and can only be run on Dormouse server. The high-level language, Dog, seems simple, clear and expressive, but still there is still a learning curve based on the fact that not all crowdsourcing requesters are developers. This is the issue for the command-line tool as well.

Another important limitation is lack of progress idiom within the system. Jabberwocky receives script definitions (deployment to Dormouse server), a process starts. Once all of them are completed, the output is written to a destination file and process terminates. In that respect, there is no way that end user can observe the current state of problem-solving.

Even if this work aims at solving general-purpose and real-world problems, it is mentioned that real-time and single-worker sequential applications are not well-suited. Despite MapReduce paradigm is simple and many social computing problems fit naturally to this paradigm, it is obvious that only some set of problems are appropriate to be solved using such a paradigm or system, which is another limitation.

However, Jabberwocky's notion of real identity and social structure, which would allow end users to define person-level constraints, is noteworthy.

6.1.2 WeFlow

WeFlow [10] is a collaborative application specification, application generator and execution engine proposed in a master thesis [35]. A framework is introduced to create and run collaboration-based applications. That framework allows end users to decompose problem into tasks, describe the computation resources, define the control and data flow.

A task consists of input(s), instruction describing the expected action from user and output(s). Task can be a type of basic (the most simple, atomic work definition), conditional (execution based on a condition), repetition (recurring execution based on a conditional), doall (groups of tasks executed in parallel) or collective (multiple times execution). The framework is definitely restricted by these predefined task definitions.

Besides task specification, end user should designate the control flow and data flow. Control flow determines the order of tasks to-be-executed. On the other hand, data flow is defined to map data between and/or within tasks. All these

specifications are done via XML. Despite the fact that the framework is mentioned to require no programming skills and XML is widely used format for representing arbitrary data structures, there is a learning process of WeFlow specifications not to mention the lack of development and debugging environments.

WeFlow differentiates human workers from each other depending on a role definition. Participants are linked to some role. Likewise a task is associated with a role. Although the whole role definition would allow end user handle access control by describing permission-like roles, it does not discriminate human performers based on their characteristics such as age, gender, interest, expertise etc.

6.1.3 TurKit

TurKit [11] is a toolkit for deploying iterative tasks to MTurk. Toolkit is based on a model that concentrate on iterative work in which series of individuals work on tasks that are previously completed by others. Although creators of TurKit apply several nontrivial problems (image description, brainstorming, writing tasks, sorting etc.) to the iterative model, this work majorly restricted by the iterative paradigm that toolkit operates on. The complex and sophisticated problems that are expected to be addressed by crowdsourcing systems do not often correspond to iterative model.

TurKit API is defined to help writing iterative MTurk tasks. However, TurKit expects end user to be a programmer and create HTML pages for tasks, and write Javascript files using API. In fact, end user is responsible for testing and making sure that the pages with TurKit functionality interact properly with MTurk. This just reveals another major limitation of TurKit on it's usefulness.

However, the toolkit have some notion of fault tolerance preventing wasted money or time due to bugs and system crashes. Toolkit stores information about the trace of a program's execution. This trace is used whenever program crashes and to put the program back into it's previous state. Thus, toolkit does not

re-execute the whole program, but the sections where they are left unfinished.

6.1.4 CrowdLang

CrowdLang [12] is a general-purpose framework and a concept of an executable, model-based programming language for workflow definition. The framework is developed based on the assumption that a complex problem is characterized by defining the problem, decomposing the problem into subproblems, planning subproblems, executing the plan and aggregating the solutions of subproblems.

CrowdLang programming language is based on a small set of operators: Divide-and-Conquer, Aggregate, Multiply, Merge, Router and Reduce (functionality of operators can be understood by their names). These operators are combined together to solve complex problems by routing, distributing and task decomposition. In addition, different types of genes are defined to address various participation patterns.

This work introduces a good and novel concept for general-purpose crowdsourcing that is suitable to most problems. The framework is not bounded to some other programming paradigm or limited to only one aspect of crowdsourcing. It rather supports complex coordination mechanisms. In this respect, translation, which is a sophisticated and difficult problem for crowdsourcing, is attempted in [24] and the results (translation of 15 different articles in less an hour) are promising.

6.1.5 AutoMan

AutoMan [13] is described to be a fully automatic crowd programming s ystem. It is a programming system integrating human and computer computation. On top of AutoMan system, a domain specific programming language is defined. It is implemented as embedded domain-specific language for Scala.

The system's whole crowdsourcing concept is based on Question objects.

Question can be type of radio-button, checkbox and restricted free-text. The human computation aspect of system only corresponds to the answers given by individuals. Further, system provides no mechanism to design complex problems through a set of subproblems or tasks. This makes system no advantageous to the existing systems depending on simple tasks.

In fact, end users are supposed to write programs in AutoMan DSL and provide them to the system. Considering the examples in the paper, the learning curve for this language can be expected to be high, because it requires knowledge of Scala language (comparing with Dog introduced in [9]).

However, scheduling, pricing and quality control mechanisms are significant components of AutoMan. The runtime system has a scheduling component that periodically checks the current situation of the results, and reprices and restarts human tasks as necessary. By making this, system tries to achieve the predefined confidence level (by end user) while staying under budget.

6.1.6 Turkomatic

Turkomatic [14] is a tool that recruits workers for planning and solving complex tasks. The system executes price-divide-solve approach by asking workers to divide complex steps into simpler ones until they are at a simple level, then to solve them. The approach simply uses divide-and-conquer strategy, but the division is done by the crowd different from other crowdsourcing systems.

The system has a set of pre-structured task templates. End users can produce workflows by combining templates together without implementing any software or designing intermediate tasks. The price-divide-solve approach is expected to produce a directed, acyclic task graph in which nodes represent subtasks and links describe task dependencies. The user interface visualizes the task graph and enables endusers delete or modify them in real-time.

The system is significant by demonstrating complex problems through acyclic task graphs. The graphs are not just shown to the endusers, but also implemented

in a way that enables real-time modification. However, initial workflow design is left to workers by a single instruction (a few sentences) telling what the enduser achieve in the end. This approach is limited by the efficiency of instruction and understanding of workers, and reportedly not really successful. It is mentioned that for complex work manual intervention and editing of workflow is effective.

Currently the system's crowd planned workflows are not guaranteed to be context free. This restricts the set of problems that the system can tackle, but still this study presents promising results by demonstrating and managing complex problems through acyclic graphs.

6.1.7 CrowdForge

CrowdForge [4] is a general-purpose framework and toolkit for accomplishing complex and interdependent tasks using micro-task markets. The framework is built on MapReduce [34] approach, which first breaks down a complex problem into a sequence of subproblems, and combines the results later. A similar approach is taken for the design of ManReduce in Jabberwocky stack [9].

Although CrowdForge approach is designed to fulfill complex problems, it is still limited to the capabilities of MapReduce approach. Some problems may not be addressed by this approach such as the case for tasks that cannot be really decomposed. Another case would be when subtasks are not independent, but the state or result of one is important to complete the other. All these exemplifies the limitations of the approach taken from distributed computing field and expected to fit well in crowdsourcing.

Additionally system has no support for iteration or recursion. It requires the end user (task designer) to specify each stage (partition, map, reduce) in the task flow.

6.1.8 CrowdWeaver

CrowdWeaver [15] is a system to visually manage crowd work. This system comes forward among various other works with it's visual representation abilities. High-level representation of a workflow makes it easier for end user to grasp, design and develop crowdsourcing programs.

The system has a predefined set of task templates. Workflows are created by creating tasks and linking them with each other in various ways. Branching and combining multiple data flows are supported and that makes design of complex problems possible.

Besides visualization, CrowdWeaver has a notion of tracking and notification. The task progress component monitors the current state and depicts the current state via graphs. Along with notification component, users can be notified about the progress based on the predefined conditions (specified by users). Further, it is possible to stop a task and make changes on existing branch in real-time.

Despite CrowdWeaver demonstrates a system that can be easily utilized by users with no programming background, the system is currently limited by predefined set of templates. In fact, it's visual abilities are restricted by only showing the workflow, but not enabling users make changes on it directly.

Feature	Jabberwocky	WeFlow	TurKit	CrowdLang	AutoMan	Turkomatic	CrowdForge	CrowdWeaver	**Crowdy**
Concept	MapReduce	Workflow	Iterative	N/A	N/A	Divide&Conquer	MapReduce	Data Flow	Stream
Programming	●	○	●	●	●	○	○	○	○
User Interface	○	○	○	○	○	●	●	●	●
Human Res.	●	●	●	●	●	●	●	●	●
Software Res.	●	●	○	●	●	○	○	●	●
Heterogeneity	●	◐	○	○	○	○	○	○	●
Quality Control	○	○	◐	○	◐	○	○	◐	●

●: Yes
◐: Partially
○: No

Table 6.1: Comparison of existing crowdsourcing platforms.

6.2 Other Studies

6.2.1 Soylent

Soylent [17] is a word processing interface enabling Microsoft Word users to employ MTurk workers to shorten, proofread and edit parts of their documents on demand. The system is developed with respect to the Find-Fix-Verify pattern that is introduced in the same study.

The pattern approaches complex tasks (focused on text editing) by splitting them into a series of generation and review stages. Rather than asking a single worker to read and edit an entire section, the pattern first recruits a set of workers to find areas of improvements. Then, another set of workers review the candidate areas and filter out incorrect ones. Finally, in the verify stage workers perform quality control on previous submissions. Throughout the process the pattern utilizes independent agreement and voting.

Both the system and the pattern concentrates on a small set of use cases of crowdsourcing. The system can only be employed by Microsoft Word users for editing text. The pattern is limited to the problems where decomposition into subproblems is possible.

6.2.2 Qurk

Qurk [16, 18] is query processing system that automatically translates queries into tasks to-be-completed by humans. The system has domain-specific language to express tasks. A UDF-like approach is taken by integrating SQL with MTurk expressions.

The approach is concentrated on a MTurk-aware database system rather than a general-purpose crowdsourcing mechanism. Thus, the set of tasks that can be introduced via the system is limited by associated database concepts such as joining, sorting etc. Although generative tasks for which workers give unconstrained

input are made available, still processing is done on input data obtained from the underlying database system. However, this study provides an important example by employing people in a database system for managing various queries.

6.2.3 Mobi

Mobi [20] is a system to crowdsource itinerary plans. The system illustrates a use case of crowdware paradigm. The paradigm focuses on tasks with global requirements and provides a single workspace in which a crowd of individuals contribute opportunistically to the global solution based on their knowledge and expertise. Itinerary planning is taken as a case study and implemented in Mobi system through a single interface.

In the system and paradigm, the problem definition is limited to the tasks with global constraints. This is a clear example of a study that concentrates on only one aspect of crowdsourcing. However, the study provides insights on the effectiveness of using unified solution context for workers or directing crowd's submissions through a structured guide or benefits of iterative refinements.

6.2.4 CrowdSpace

CrowdSpace [19] is a system that supports the evaluation of complex crowd work by combining information about worker results through visualization. This system focuses on exploration and assessment of worker performance and behavior rather than providing ways to manage complex workflows.

The system presents a unified user interface with four components: a scatter plot of aggregate behavioral features (time spent on the task, number of keys pressed while processing the task etc.), distribution of each behavioral features, traces of worker/output pairs and overall worker behavior based on their answers.

Low quality work is common in crowdsourcing. However, this system provides great insights by combining worker behavior with their submissions, and enabling

end users to identify the behavior of workers who have good or bad outputs. Although this approach is limited by several aspects such as being only applicable to pages in which Javascript can be inserted or assuming that worker does all the processing on the page etc, the quality control approach taken in the study can be used to better understand and address the nature of the crowd.

6.2.5 Human Architecture

Human Architecture [21] is an adaptation mechanism to the changing requirements of the various type collaborations. Unlike other studies, this work focuses on collaboration problem from the architectural perspective. An architecture description language is proposed to describe collaboration dynamics. Considering software architecture human components and collaboration connectors are introduced to demonstrate coordination dependencies.

However, the proposed human architecture approach is too architecture focused and highly complex. The collaboration of individuals is narrowed down to component and connectors, but in the context of complex problems collaboration is dynamically changing and highly interactive due to the data items that are output from one and input to the other.

Chapter 7

Discussion

Crowdy fundamentally depends on the idea that real-world problems can be demonstrated using component-based model. This dependency requires that a complex problem can be broken into pieces of smaller tasks and these tasks can be coordinated to solve the overall problem. However, there may be cases when this assumption can be violated. It is possible to have some work that may not be easily divided into smaller unit of work. For example, asking people's ideas on several topics cannot be easily divided into several tasks. However, this case can be still tackled by *Crowdy* application by having a source operator and a sink operator in which the data tuples containing various ideas generated by human workers via human source operator are written to a file by sink file operator or emailed to the requester by sink email operator. It is still possible that the decomposition of problem into various tasks and the implementation of those tasks over operators along with necessary quality control steps could introduce some overhead and cost. Nevertheless, the premise of *Crowdy* is still valid by enabling human-computer collaboration over a component-based model without going through manual processes.

Secondly, the existing platform does not fully support iteration. Although specific set of operators (human operator, enrich operator) have iteration support using that a task can be copied and processed for a number of times without a need to copy the actual operator, an output of an operator cannot input another

operator that is up in the flow. In other words, loops are not allowed. This limitation is due to the fact that data tuples flow through the application from source to sink operators. The full iteration support is left open for the future discussions and improvements on the platform.

Another limitation is the scope of human collaboration. *Crowdy* supports collaboration of tasks that are completed individually by human workers. The collaborative task completion for which more than one person work on a task together in real-time is not supported. Real-time collaboration is beyond the scope of this work. However, a task or a piece of work can be completed by more than one person. Output of a human task can be taken and input to another human task. In this way, people collaborate with each other to complete a piece of work, but this is not happening in real-time. Even though interdependency of tasks can be easily handled by *Crowdy*, human workers should be provided with enough context and information on the problem that they are collaborating to achieve the best results.

Quality control is the most important challenge for crowdsourcing systems. Since low quality submissions for human tasks is common, most of the quality control efforts focus on human-related computation. Currently quality control can be achieved by adding extra human operators in which we can employ other people to check whether a completed human task has a good quality or not. However, the low quality submissions are still possible for the human tasks to evaluate quality of work done by others. Possible improvement aspects on this issue is further discussed in Chapter 8.

Chapter 8

Conclusion

In this work, an extensible and general-purpose crowdsourcing framework is defined and a platform is developed to solve sophisticated problems. *Crowdy* can benefit users (requesters or task designers) through easier and more efficient management of collaboration in between resources. It allows requesters to describe and implement a problem with no requirement of programming knowledge.

The framework allows users to define real-world problems using a component-based model and implement a solution by creating a crowdsourcing application with ease. Platform manages the coordination between computer and human components effectively and produces the results that user is asking for. Based on the concepts of stream processing, *Crowdy* provides an efficient way to describe problems by employing various components and managing the flow of information and dependencies between them. Two case studies and multiple experiments show how the component-based framework can handle complex and sophisticated problems and lead solutions to solve them. In these studies, it is proved that the platform is capable of not just finding a solution to the problem, but it can solve the problem in several different ways.

8.1 Future Work

During the development of the actual platform and tests over different scenarios the following observations and enhancements have been identified. These improvements and ideas listed in this section conforms to a promising future work for the existing platform.

8.1.1 Extending the Operator Set

Existing platform provides a set of operators to define and solve problems. This set is proved to be enough for given real-world scenarios such as translation or finding business addresses. However, the problem set can be expanded by various other scenarios and problems. Therefore, the existing operator set can be extended by several types of operators. For example, social media has become the main communication method to share and exchange information and ideas online. Therefore, it is inevitable to expect social media related computations such as classifying tweets. Let's assume that user needs a crowdsourcing application that classifies and saves tweets into a file with respect to their sentiment identified by human workers. The current platform is capable to handle such a scenario as follows. A list of tweets is input to the application via *source manual operator*. Human workers analyze each tweet and identify a sentiment via *human processing operator*, which has an input port to receive tweets. The identified tweets are saved into a file by their class (sentiment) that is split *split operator*. In this scenario, *source manual operator* can be replaced by a source operator that is capable of reading Twitter API to-be-configured by specific set of parameters.

Besides Twitter API reader kind of specific functionality operators, the existing operator set can be further extended. In terms of source operators, file readers and RSS readers might be useful for certain scenarios such as reading content from files or web pages directly rather than using source manual operator. Having a projection relational operator can be another enhancement for the existing system, although not having that operator does not impact the system

functionality significantly.

Considering current platform definition, one crucial improvement might be adding feedback ports to certain set of operators in addition to input and output ports. This feedback port can be activate and deactivate an operator with respect to a given condition. Using this operator the crowdsourcing application gains more dynamism and ability to change internal functional details according to the conditions emerged by given input set.

8.1.2 Improvements to Existing Operators

The existing set of operators can be further improved individually. However, most of these improvements needs thorough testing and analysis. Here is a list of refinement that can be applied to existing operators:

human operator. Requester can be enabled to change instructions and question design by changing format, adding/removing image/audio/video or basically adding HTML.

Currently requester can create four types input to be completed by human workers: text input, number input, single selection or multiple selection. This list can be extended by other types of questions.

Heterogeneity of human operators is a fundamental aspect of human computation. Right now the platform provides a set of rules to the requester to assign task to the right worker. This can be further improved by allowing requesters to create their own custom rules to test whether a worker is qualified to complete that task or not.

Further, human operators can be possibly integrated into other task markets and crowdsourcing services. Although this work utilizes Amazon's MTurk as a resource for human computation, there are other available services such as CrowdFlower or oDesk that can be applied. If it is possible to apply the platform to the service where the details of each individual worker better known, that

might lead better and greater opportunities to manage resource allocation and task assignment.

source manual operator. Source manual operator can be further enriched by adding more delimiter options.

sink email operator. Sink email operator can be configured to have parameters for subject and/or body.

sink file operator. Sink file operator writes data tuples into a tab delimited file. This delimiter can be set by requesters.

selection/split operators. Considering selection and split operators, more boolean predicates (greater than, less than etc.) can be added to test whether a data tuple evaluates to true or false.

8.1.3 More Quality Control

The quality control of the current platform definition can be additionally enhanced by having operators dedicated to quality control. These operators can be used to check various parameters of the task completed by human workers such as the duration takes to complete the task, the time human worker spends on the task page, the length of the submission in terms of characters, the number of key strokes occur while human worker is working on the task, the movements of the mouse etc. This approach requires a detailed analysis on the possible list of parameters. Further, a concrete evaluation criteria of a parameter list is significantly important, and this requirement not only considers computer science studies, but it necessary to bring efforts from different domains (sociology, business information etc.) to come up with a right set of definitions and rules for quality control.

Bibliography

[1] R. P. Gabriel, L. Northrop, D. C. Schmidt, and K. Sullivan, "Ultra-large-scale systems," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 632–634, ACM, 2006.

[2] D. G. Messerschmitt and C. Szyperski, "Software ecosystem: understanding an indispensable technology and industry," *MIT Press Books*, vol. 1, 2005.

[3] C. Dorn and R. N. Taylor, "Analyzing runtime adaptability of collaboration patterns," in *Collaboration Technologies and Systems (CTS), 2012 International Conference on*, pp. 551–558, IEEE, 2012.

[4] A. Kittur, B. Smus, S. Khamkar, and R. E. Kraut, "Crowdforge: Crowdsourcing complex work," in *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pp. 43–52, ACM, 2011.

[5] A. Doan, R. Ramakrishnan, and A. Y. Halevy, "Crowdsourcing systems on the world-wide web," *Communications of the ACM*, vol. 54, no. 4, pp. 86–96, 2011.

[6] H. Zhang, E. Horvitz, R. C. Miller, and D. C. Parkes, "Crowdsourcing general computation," 2011.

[7] A. Kittur, J. V. Nickerson, M. Bernstein, E. Gerber, A. Shaw, J. Zimmerman, M. Lease, and J. Horton, "The future of crowd work," in *Proceedings of the 2013 conference on Computer supported cooperative work*, pp. 1301–1318, ACM, 2013.

[8] A. Bernstein, M. Klein, and T. W. Malone, "Programming the global brain," *Communications of the ACM*, vol. 55, no. 5, pp. 41–43, 2012.

[9] S. Ahmad, A. Battle, Z. Malkani, and S. Kamvar, "The jabberwocky programming environment for structured social computing," in *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pp. 53–64, ACM, 2011.

[10] N. Kokciyan, S. Uskudarli, and T. Dinesh, "User generated human computation applications," in *Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom)*, pp. 593–598, IEEE, 2012.

[11] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller, "Turkit: tools for iterative tasks on mechanical turk," in *Proceedings of the ACM SIGKDD workshop on human computation*, pp. 29–30, ACM, 2009.

[12] P. Minder and A. Bernstein, "Crowdlang-first steps towards programmable human computers for general computation.," in *Human Computation*, 2011.

[13] D. W. Barowy, C. Curtsinger, E. D. Berger, and A. McGregor, "Automan: A platform for integrating human-based and digital computation," in *ACM SIGPLAN Notices*, vol. 47, pp. 639–654, ACM, 2012.

[14] A. Kulkarni, M. Can, and B. Hartmann, "Collaboratively crowdsourcing workflows with turkomatic," in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pp. 1003–1012, ACM, 2012.

[15] A. Kittur, S. Khamkar, P. André, and R. Kraut, "Crowdweaver: visually managing complex crowd work," in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pp. 1033–1036, ACM, 2012.

[16] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller, "Human-powered sorts and joins," *Proceedings of the VLDB Endowment*, vol. 5, no. 1, pp. 13–24, 2011.

[17] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich, "Soylent: a word processor with

a crowd inside," in *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, pp. 313–322, ACM, 2010.

[18] A. Marcus, E. Wu, D. R. Karger, S. R. Madden, and R. C. Miller, "Crowdsourced databases: Query processing with people," CIDR, 2011.

[19] J. Rzeszotarski and A. Kittur, "Crowdscape: interactively visualizing user behavior and output," in *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pp. 55–62, ACM, 2012.

[20] H. Zhang, E. Law, R. Miller, K. Gajos, D. Parkes, and E. Horvitz, "Human computation tasks with global constraints," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 217–226, ACM, 2012.

[21] C. Dorn and R. N. Taylor, "Architecture-driven modeling of adaptive collaboration structures in large-scale social web applications," in *Web Information Systems Engineering-WISE 2012*, pp. 143–156, Springer, 2012.

[22] D. Schall, S. Dustdar, and M. B. Blake, "Programming human and software-based web services," *Computer*, pp. 82–85, 2010.

[23] C. Dorn, R. N. Taylor, and S. Dustdar, "Flexible social workflows: Collaborations as human architecture," *Internet Computing, IEEE*, vol. 16, no. 2, pp. 72–77, 2012.

[24] P. Minder and A. Bernstein, "How to translate a book within an hour: towards general purpose programmable human computers with crowdlang," in *Proceedings of the 3rd Annual ACM Web Science Conference*, pp. 209–212, ACM, 2012.

[25] A. J. Quinn and B. B. Bederson, "A taxonomy of distributed human computation," *Human-Computer Interaction Lab Tech Report, University of Maryland*, 2009.

[26] A. J. Quinn and B. B. Bederson, "Human computation: a survey and taxonomy of a growing field," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 1403–1412, ACM, 2011.

[27] J. Howe, "The rise of crowdsourcing," *Wired magazine*, vol. 14, no. 6, pp. 1–4, 2006.

[28] J. Howe, "Crowdsourcing: A definition," 2006. [Online].

[29] P. G. Ipeirotis, "Analyzing the amazon mechanical turk marketplace," *XRDS: Crossroads, The ACM Magazine for Students*, vol. 17, no. 2, pp. 16–21, 2010.

[30] C. Callison-Burch, "Fast, cheap, and creative: evaluating translation quality using amazon's mechanical turk," in *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*, pp. 286–295, Association for Computational Linguistics, 2009.

[31] H. Andrade, B. Gedik, and D. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014.

[32] L. Richardson and S. Ruby, *RESTful web services*. O'Reilly Media, Inc., 2008.

[33] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little, *Documenting software architectures: views and beyond*. Pearson Education, 2002.

[34] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[35] N. Kokciyan, "Weflow: We follow the flow," Master's thesis, Galatasaray University, 2009.