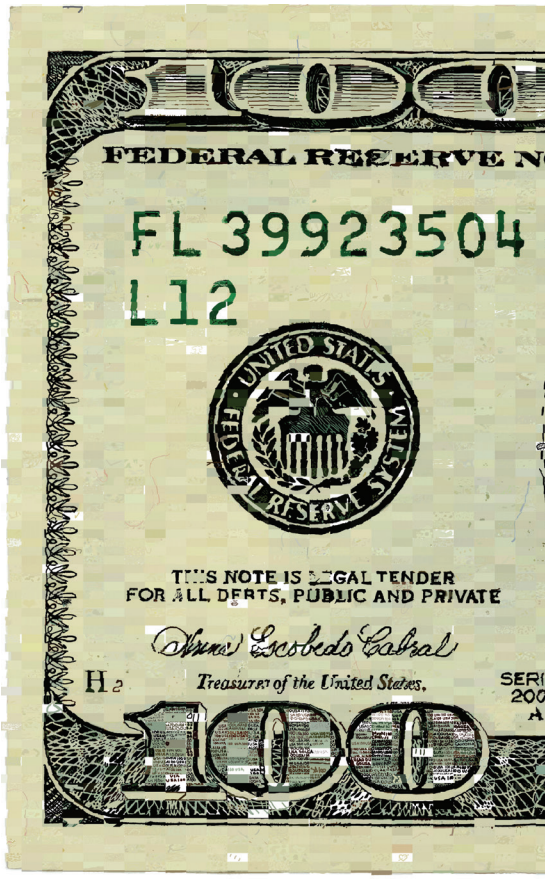
DOI:10.1145/1924421.1924442

The practice of crowdsourcing is transforming the Web and giving rise to a new field.

BY ANHAI DOAN, RAGHU RAMAKRISHNAN, AND ALON Y. HALEVY

Crowdsourcing Systems on the World-Wide Web



CROWDSOURCING SYSTEMS enlist a multitude of humans to help solve a wide variety of problems. Over the past decade, numerous such systems have appeared on the World-Wide Web. Prime examples include Wikipedia, Linux, Yahoo! Answers, Mechanical Turk-based systems, and much effort is being directed toward developing many more.

As is typical for an emerging area, this effort has appeared under many names, including peer production, user-powered systems, user-generated content, collaborative systems, community systems, social systems, social search, social media, collective intelligence, wikinomics, crowd wisdom, smart mobs, mass collaboration, and human computation. The topic has been discussed extensively in books, popular press, and academia.[1,5,15,23,29,35] But this body of work has considered mostly efforts in the physical world.[23,29,30] Some do consider crowdsourcing systems on the Web, but only certain system types[28,33] or challenges (for example, how to evaluate users[12]).

This survey attempts to provide a global picture of crowdsourcing systems on the Web. We define and classify such systems, then describe a broad sample of systems. The sample

» key insights

- **Crowdsourcing systems face four key challenges: How to recruit contributors, what they can do, how to combine their contributions, and how to manage abuse. Many systems "in the wild" must also carefully balance openness with quality.**
- **The race is on to build general crowdsourcing platforms that can be used to quickly build crowdsourcing applications in many domains. Using these, we can already build databases previously unimaginable at lightning speed.**

Ten Thousand Cents is a digital artwork by Aaron Koblin that creates a representation of a $100 bill. Using a custom drawing tool, thousands of individuals, working in isolation from one another, painted a tiny part of the bill without knowledge of the overall task.

ARTWORK BY AARON KOBLIN AND TAKASHI KAWASHIMA

ranges from relatively simple well-established systems such as reviewing books to complex emerging systems that build structured knowledge bases to systems that "piggyback" onto other popular systems. We discuss fundamental challenges such as how to recruit and evaluate users, and to merge their contributions. Given the space limitation, we do not attempt to be exhaustive. Rather, we sketch only the most important aspects of the global picture, using real-world examples. The goal is to further our collective understanding—both conceptual and practical—of this important emerging topic.

It is also important to note that many crowdsourcing platforms have been built. Examples include Mechanical Turk, Turkit, Mob4hire, uTest, Freelancer, eLance, oDesk, Guru, Topcoder, Trada, 99design, Innocentive, CloudCrowd, and CloudFlower. Using these platforms, we can quickly build crowdsourcing systems in many domains. In this survey, we consider these systems (that is, applications), not the crowdsourcing platforms themselves.

Crowdsourcing Systems

Defining crowdsourcing (CS) systems turns out to be surprisingly tricky. Since many view Wikipedia and Linux as well-known CS examples, as a natural starting point, we can say that a CS system enlists a crowd of users to *explicitly* collaborate to build a long-lasting *artifact* that is beneficial to the whole community.

This definition, however, appears too restricted. It excludes, for example, the ESP game,[32] where users *implicitly* collaborate to label images as a side effect while playing the game. ESP clearly benefits from a crowd of users. More importantly, it faces the same human-centric challenges of Wikipedia and Linux, such as how to recruit and evaluate users, and to combine their contributions. Given this, it seems unsatisfactory to consider only explicit collaborations; we ought to allow implicit ones as well.

The definition also excludes, for example, an Amazon's Mechanical Turk-based system that enlists users to find a missing boat in thousands of satellite images.[18] Here, users do not build any artifact, arguably nothing is long lasting, and no community exists either (just users coming together for this particular task). And yet, like ESP, this system clearly benefits from users, and faces similar human-centric challenges. Given this, it ought to be considered a CS system, and the goal of building artifacts ought to be relaxed into the more general goal of solving problems. Indeed, it appears that in principle *any* non-trivial problem *can* benefit from crowdsourcing: we can describe the problem on the Web, solicit user inputs, and examine the inputs to develop a solution. This system may not be practical (and better systems may exist), but it can arguably be considered a primitive CS system.

Consequently, we do not restrict the type of collaboration nor the target problem. Rather, we view CS as a general-purpose problem-solving method. We say that a system is a CS system *if it enlists a crowd of humans to help solve a problem defined by the system owners*, and if in doing so, it addresses the following four fundamental challenges:

A sample of basic CS system types on the World-Wide Web.

Nature of Collaboration	Architecture	Must recruit users?	What users do?	Examples	Target Problems	Comments
Explicit	Standalone	Yes	Evaluating ▸ review, vote, tag	▸ reviewing and voting at Amazon, tagging Web pages at del.ici.ous.com and Google Co-op	Evaluating a collection of items (e.g., products, users)	Humans as perspective providers. No or loose combination of inputs.
			Sharing ▸ items ▸ textual knowledge ▸ structured knowledge	▸ Napster, YouTube, Flickr, CPAN, programmableweb.com ▸ Mailing lists, Yahoo! Answers, QUIQ, ehow.com, Quora ▸ Swivel, Many Eyes, Google Fusion Tables, Google Base, bmrb.wisc.edu, galaxyzoo, Piazza, Orchestra	Building a (distributed or central) collection of items that can be shared among users.	Humans as content providers. No or loose combination of inputs.
			Networking	▸ LinkedIn, MySpace, Facebook	Building social networks	Humans as component providers. Loose combination of inputs.
			Building artifacts ▸ software ▸ textual knowledge bases ▸ structured knowledge bases ▸ systems ▸ others	▸ Linux, Apache, Hadoop ▸ Wikipedia, openmind, Intellipedia, ecolicommunity ▸ Wikipedia infoboxes/DBpedia, IWP, Google Fusion Tables, YAGO-NAGA, Cimple/DBLife ▸ Wikia Search, mahalo, Freebase, Eurekster ▸ newspaper at Digg.com, Second Life	Building physical artifacts	Humans can play all roles. Typically tight combination of inputs. Some systems ask both humans and machines to contribute.
			Task execution	▸ Finding extraterrestrials, elections, finding people, content creation (e.g., Demand Media, Associated Content)	Possibly any problem	
Implicit	Standalone	Yes	▸ play games with a purpose ▸ bet on prediction markets ▸ use private accounts ▸ solve captchas ▸ buy/sell/auction, play massive multiplayer games	▸ ESP ▸ intrade.com, Iowa Electronic Markets ▸ IMDB private accounts ▸ recaptcha.net ▸ eBay, *World of Warcraft*	▸ labeling images ▸ predicting events ▸ rating movies ▸ digitizing written text ▸ building a user community (for purposes such as charging fees, advertising)	Humans can play all roles. Input combination can be loose or tight.
	Piggyback on another system	No	▸ keyword search ▸ buy products ▸ browse Web sites	▸ Google, Microsoft, Yahoo ▸ recommendation feature of Amazon ▸ adaptive Web sites (e.g., Yahoo! front page)	▸ spelling correction, epidemic prediction ▸ recommending products ▸ reorganizing a Web site for better access	Humans can play all roles. Input combination can be loose or tight.

How to recruit and retain users? What contributions can users make? How to combine user contributions to solve the target problem? How to evaluate users and their contributions?

Not all human-centric systems address these challenges. Consider a system that manages car traffic in Madison, WI. Its goal is to, say, coordinate the behaviors of a crowd of human drivers (that already exist *within* the system) in order to minimize traffic jams. Clearly, this system does not want to recruit more human drivers (in fact, it wants far fewer of them). We call such systems *crowd management (CM) systems*. CM techniques (a.k.a., "crowd coordination"[31]) can be relevant to CS contexts. But the two system classes are clearly distinct.

In this survey we focus on CS systems that leverage the Web to solve the four challenges mentioned here (or a significant subset of them). The Web is unique in that it can help recruit a large number of users, enable a high degree of automation, and provide a large set of social software (for example, email, wiki, discussion group, blogging, and tagging) that CS systems can use to manage their users. As such, compared to the physical world, the Web can dramatically improve existing CS systems and give birth to novel system types.

Classifying CS systems. CS systems can be classified along many dimensions. Here, we discuss nine dimensions we consider most important. The two that immediately come to mind are the *nature of collaboration* and *type of target problem*. As discussed previously, collaboration can be explicit or implicit, and the target problem can be any problem defined by the system owners (for example, building temporary or permanent artifacts, executing tasks).

The next four dimensions refer respectively to how a CS system solves the four fundamental challenges described earlier: *how to recruit and retain users; what can users do; how to combine*

their inputs; and how to evaluate them. Later, we will discuss these challenges and the corresponding dimensions in detail. Here, we discuss the remaining three dimensions: degree of manual effort, role of human users, and standalone versus piggyback architectures.

Degree of manual effort. When building a CS system, we must decide how much manual effort is required to solve each of the four CS challenges. This can range from relatively little (for example, combining ratings) to substantial (for example, combining code), and clearly also depends on how much the system is automated. We must decide how to divide the manual effort between the users and the system owners. Some systems ask the users to do relatively little and the owners a great deal. For example, to detect malicious users, the users may simply click a button to report suspicious behaviors, whereas the owners must carefully examine all relevant evidence to determine if a user is indeed malicious. Some systems do the reverse. For example, most of the manual burden of merging Wikipedia edits falls on the users (who are currently editing), not the owners.

Role of human users. We consider four basic roles of humans in a CS system. *Slaves*: humans help solve the problem in a divide-and-conquer fashion, to minimize the resources (for example, time, effort) of the owners. Examples are ESP and finding a missing boat in satellite images using Mechanical Turk. *Perspective providers*: humans contribute different perspectives, which when combined often produce a better solution (than with a single human). Examples are reviewing books and aggregating user bets to make predictions.[29] *Content providers*: humans contribute self-generated content (for example, videos on YouTube, images on Flickr). *Component providers*: humans function as components in the target artifact, such as a social network, or simply just a community of users (so that the owner can, say, sell ads). Humans often play multiple roles within a single CS system (for example, slaves, perspective providers, and content providers in Wikipedia). It is important to know these roles because that may determine how to recruit. For example, to use humans as perspective providers, it is important to recruit a diverse crowd where each human can make independent decisions, to avoid "group think."[29]

Compared to the physical world, the Web can dramatically improve existing crowdsourcing systems and give birth to novel system types.

Standalone versus piggyback. When building a CS system, we may decide to piggyback on a well-established system, by exploiting traces that users leave in that system to solve our target problem. For example, Google's "Did you mean" and Yahoo's Search Assist utilize the search log and user clicks of a search engine to correct spelling mistakes. Another system may exploit user purchases in an online bookstore (Amazon) to recommend books. Unlike standalone systems, such piggyback systems do not have to solve the challenges of recruiting users and deciding what they can do. But they still have to decide how to evaluate users and their inputs (such as traces in this case), and to combine such inputs to solve the target problem.

Sample CS Systems on the Web

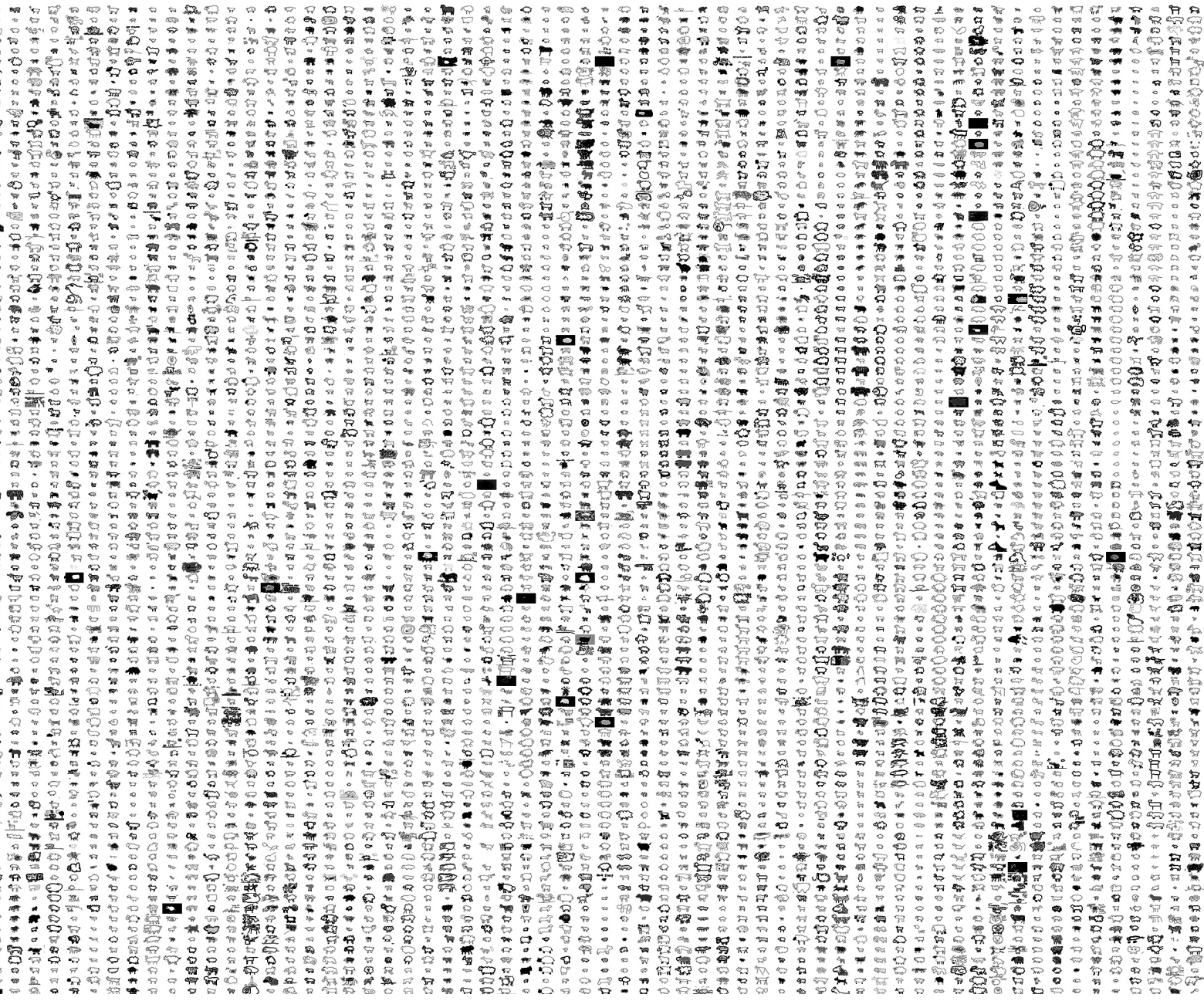
Building on this discussion of CS dimensions, we now focus on CS systems on the Web, first describing a set of basic system types, and then showing how deployed CS systems often combine multiple such types.

The accompanying table shows a set of basic CS system types. The set is not meant to be exhaustive; it shows only those types that have received most attention. From left to right, it is organized by collaboration, architecture, the need to recruit users, and then by the actions users can take. We now discuss the set, starting with explicit systems.

Explicit Systems: These standalone systems let users collaborate explicitly. In particular, users can evaluate, share, network, build artifacts, and execute tasks. We discuss these systems in turn.

Evaluating: These systems let users evaluate "items" (for example, books, movies, Web pages, other users) using textual comments, numeric scores, or tags.[10]

Sharing: These systems let users share "items" such as products, services, textual knowledge, and structured knowledge. Systems that share products and services include Napster, YouTube, CPAN, and the site programmableweb.com (for sharing files, videos, software, and mashups, respectively). Systems that share textual knowledge include mailing lists, Twitter, how-to

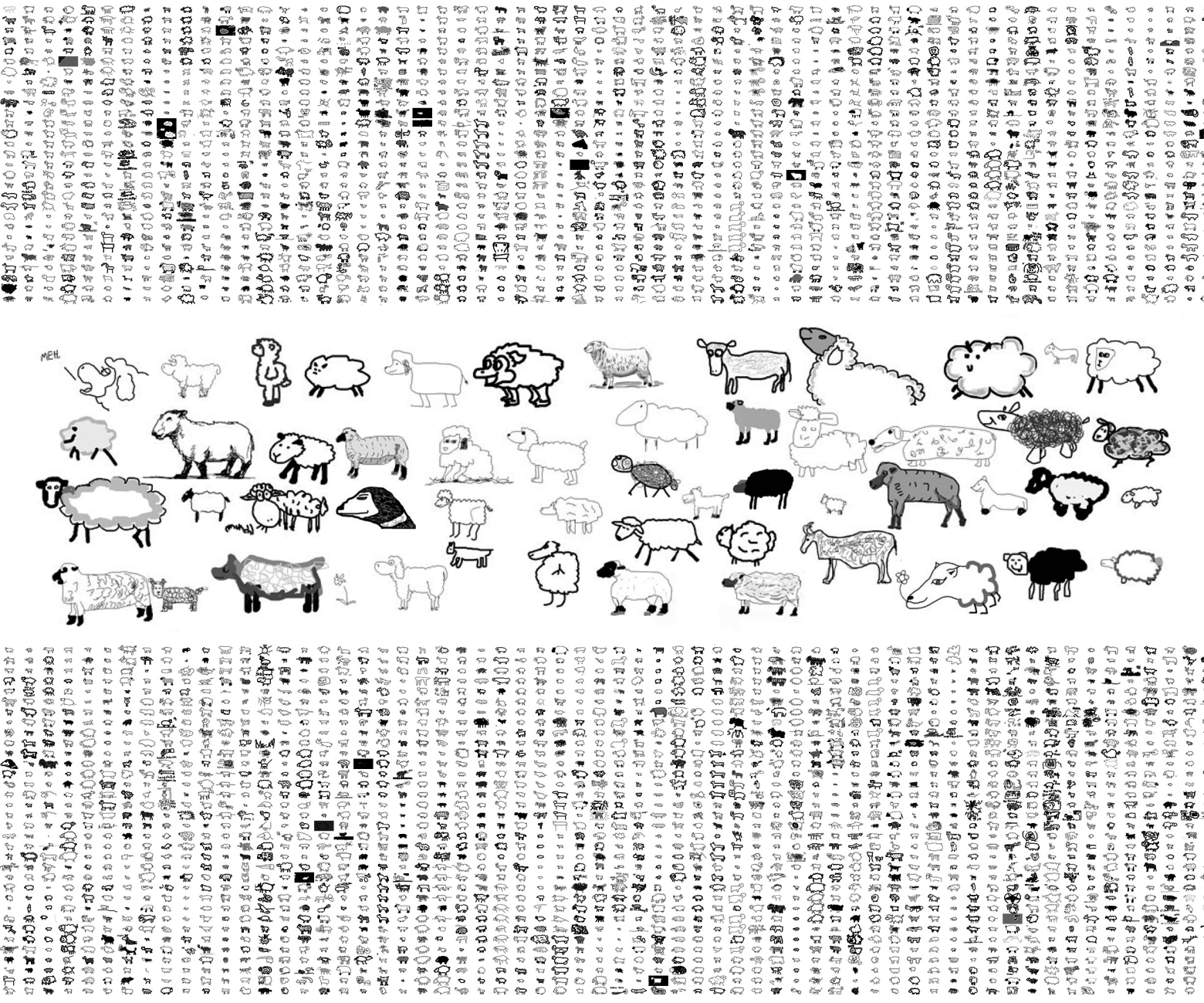


repositories (such as ehow.com, which lets users contribute and search how-to articles), Q&A Web sites (such as Yahoo! Answers[2]), online customer support systems (such as QUIQ,[22] which powered Ask Jeeves' AnswerPoint, a Yahoo! Answers-like site). Systems that share structured knowledge (for example, relational, XML, RDF data) include Swivel, Many Eyes, Google Fusion Tables, Google Base, many e-science Web sites (such as bmrb.wisc.edu, galaxyzoo.org), and many peer-to-peer systems developed in the Semantic Web, database, AI, and IR communities (such as Orchestra[8,27]). Swivel, for example, bills itself as the "YouTube of structured data," which lets users share, query, and visualize census- and voting data, among others. In general, sharing systems can be central (such as YouTube, ehow, Google Fusion Tables, Swivel) or distributed, in a peer-to-peer fashion (such as Napster, Orchestra).

Networking: These systems let users collaboratively construct a large social network graph, by adding nodes and edges over time (such as homepages, friendships). Then they exploit the graph to provide services (for example, friend updates, ads, and so on). To a lesser degree, blogging systems are also networking systems in that bloggers often link to other bloggers.

A key distinguishing aspect of systems that evaluate, share, or network is that they do not merge user inputs, or do so automatically in relatively simple fashions. For example, evaluation systems typically do not merge textual user reviews. They often merge user inputs such as movie ratings, but do so automatically using some formulas. Similarly, networking systems automatically merge user inputs by adding them as nodes and edges to a social network graph. As a result, users of such systems do not need (and, in fact, often are not allowed) to edit other users' input.

Building Artifacts: In contrast, systems that let users build artifacts such



The Sheep Market by Aaron Koblin is a collection of 10,000 sheep made by workers on Amazon's Mechanical Turk. Workers were paid $0.02 (USD) to "draw a sheep facing to the left." Animations of each sheep's creation may be viewed at TheSheepMarket.com.

ARTWORK BY AARON KOBLIN

as Wikipedia often merge user inputs tightly, and require users to edit and merge one another's inputs. A well-known artifact is software (such as Apache, Linux, Hadoop). Another popular artifact is textual knowledge bases (KBs). To build such KBs (such as Wikipedia), users contribute data such as sentences, paragraphs, Web pages, then edit and merge one another's contributions. The knowledge capture (k-cap.org) and AI communities have studied building such KBs for over a decade. A well-known early attempt is openmind,[28] which enlists volunteers to build a KB of commonsense facts (for example, "the sky is blue"). Recently, the success of Wikipedia has inspired many "community wikipedias," such as Intellipedia (for the U.S. intelligence community) and EcoliHub (at ecolicommunity.org, to capture all information about the E. coli bacterium).

Yet another popular target artifact is structured KBs. For example, the set of all Wikipedia infoboxes (that is, attribute-value pairs such as city-name = Madison, state = WI) can be viewed as a structured KB collaboratively created by Wikipedia users. Indeed, this KB has recently been extracted as DBpedia and used in several applications (see dbpedia.org). Freebase.com builds an open structured database, where users can create and populate schemas to describe topics of interest, and build collections of interlinked topics using a flexible graph model of data. As yet another example, Google Fusion Tables (tables.googlelabs.com) lets users upload tabular data and collaborate on it by merging tables from different sources, commenting on data items, and sharing visualizations on the Web.

Several recent academic projects have also studied building structured

KBs in a CS fashion. The IWP project[35] extracts structured data from the textual pages of Wikipedia, then asks users to verify the extraction accuracy. The Cimple/DBLife project[4,5] lets users correct the extracted structured data, expose it in wiki pages, then add even more textual and structured data. Thus, it builds structured "community wikipedias," whose wiki pages mix textual data with structured data (that comes from an underlying structured KB). Other related works include YAGONAGA,[11] BioPortal,[17] and many recent projects in the Web, Semantic Web, and AI communities.[1,16,36]

In general, building a structured KB often requires selecting a set of data sources, extracting structured data from them, then integrating the data (for example, matching and merging "David Smith" and "D.M. Smith"). Users can help these steps in two ways. First, they can improve the automatic algorithms of the steps (if any), by editing their code, creating more training data,[17] answering their questions[12,13] or providing feedback on their output.[12,35] Second, users can manually participate in the steps. For example, they can manually add or remove data sources, extract or integrate structured data, or add even more structured data, data not available in the current sources but judged relevant.[5] In addition, a CS system may perform inferences over its KB to infer more structured data. To help this step, users can contribute inference rules and domain knowledge.[25] During all such activities, users can naturally cross-edit and merge one another's contributions, just like in those systems that build textual KBs.

Another interesting target problem is building and improving systems running on the Web. The project Wikia Search (search.wikia.com) lets users build an open source search engine, by contributing code, suggesting URLs to crawl, and editing search result pages (for example, promoting or demoting URLs). Wikia Search was recently disbanded, but similar features (such as editing search pages) appear in other search engines (such as Google, mahalo.com). Freebase lets users create custom browsing and search systems (deployed at Freebase), using the community-curated data and a suite of development tools (such as the Metaweb query language and a hosted development environment). Eurekster.com lets users collaboratively build vertical search engines called *swickis*, by customizing a generic search engine (for example, specifying all URLs the system should crawl). Finally, MOBS, an academic project,[12,13] studies how to collaboratively build data integration systems, those that provide a uniform query interface to a set of data sources. MOBS enlists users to create a crucial system component, namely the semantic mappings (for example, "location" = "address") between the data sources.

In general, users can help build and improve a system running on the Web in several ways. First, they can edit the system's code. Second, the system typically contains a set of internal components (such as URLs to crawl, semantic mappings), and users can help improve these without even touching the system's code (such as adding new URLs, correcting mappings). Third, users can edit system inputs and outputs. In the case of a search engine, for instance, users can suggest that if someone queries for "home equity loan for seniors," the system should also suggest querying for "reverse mortgage." Users can also edit search result pages (such as promoting and demoting URLs, as mentioned earlier). Finally, users can monitor the running system and provide feedback.

We note that besides software, KBs, and systems, many other target artifacts have also been considered. Examples include community newspapers built by asking users to contribute and evaluate articles (such as Digg) and massive multi-player games that build virtual artifacts (such as *Second Life*, a 3D virtual world partly built and maintained by users).

Executing Tasks: The last type of explicit systems we consider is the kind that executes tasks. Examples include finding extraterrestrials, mining for gold, searching for missing people,[23,29,30,31] and cooperative debugging (cs.wisc.edu/cbi, early work of this project received the ACM Doctoral Dissertation Award in 2005). The 2008 election is a well-known example, where the Obama team ran a large online CS operation asking numerous volunteers to help mobilize voters. To apply CS to a task, we must find task parts that can be "crowdsourced," such that each user can make a contribution and the contributions in turn can be combined to solve the parts. Finding such parts and combining user contributions are often task specific. Crowdsourcing the parts, however, can be fairly general, and plaforms have been developed to assist that process. For example, Amazon's Mechanical Turk can help distribute pieces of a task to a crowd of users (and several recent interesting toolkits have even been developed for using Mechanical Turk[13,37]). It was used recently to search for Jim Gray, a database researcher lost at sea, by asking volunteers to examine pieces of satellite images for any sign of Jim Gray's boat.[18]

Implicit Systems: As discussed earlier, such systems let users collaborate implicitly to solve a problem of the system owners. They fall into two groups: standalone and piggyback.

A standalone system provides a service such that when using it users implicitly collaborate (as a side effect) to solve a problem. Many such systems exist, and the table here lists a few representative examples. The ESP game[32] lets users play a game of guessing common words that describe images (shown independently to each user), then uses those words to label images. Google Image Labeler builds on this game, and many other "games with a purpose" exist.[33] Prediction markets[23,29] let users bet on events (such as elections, sport events), then aggregate the bets to make predictions. The intuition is that the "collective wisdom" is often accurate (under certain conditions)[31] and that this helps incorporate inside information available from users. The Internet Movie Database (IMDB) lets users import movies into private accounts (hosted by IMDB). It designed the accounts such that users are strongly motivated to rate the imported movies, as doing so bring many private benefits (such as they can query to find all imported action movies rated at least 7/10, or the system can recommend action movies highly rated by people with similar taste). IMDB then aggregates all private ratings to obtain a public rating for each movie, for the benefit of the public. reCAPTCHA asks users to solve captchas to prove they are humans (to gain access to a site), then leverages

the results for digitizing written text.[34] Finally, it can be argued that the *target problem* of many systems (that provide user services) is simply to *grow a large community of users*, for various reasons (such as personal satisfaction, charging subscription fees, selling ads, selling the systems to other companies). Buy/sell/auction websites (such as eBay) and massive multiplayer games (such as *World of Warcraft*) for instance fit this description. Here, by simply joining the system, users can be viewed as implicitly collaborating to solve the target problem (of growing user communities).

The second kind of implicit system we consider is a piggyback system that exploits the user traces of yet another system (thus, making the users of this latter system implicitly collaborate) to solve a problem. For example, over time many piggyback CS systems have been built on top of major search engines, such as Google, Yahoo!, and Microsoft. These systems exploit the traces of search engine users (such as search logs, user clicks) for a wide range of tasks (such as spelling correction, finding synonyms, flu epidemic prediction, and keyword generation for ads[6]). Other examples include exploiting user purchases to recommend products,[26] and exploiting click logs to improve the presentation of a Web site.[19]

CS Systems on the Web

We now build on basic system types to discuss deployed CS systems on the Web. Founded on static HTML pages, the Web soon offered many interactive services. Some services serve machines (such as DNS servers, Google Map API server), but most serve humans. Many such services do not need to recruit users (in the sense that the more the better). Examples include pay-parking-ticket services (for city residents) and room-reservation services. (As noted, we call these crowd management systems). Many services, however, face CS challenges, including the need to grow large user bases. For example, online stores such as Amazon want a growing user base for their services, to maximize profits, and startups such as epinions.com grow their user bases for advertising. They started out as primitive CS systems, but quickly improved over time with additional CS features (such as reviewing, rating, networking). Then around 2003, aided by the proliferation of social software (for example, discussion groups, wiki, blog), many full-fledged CS systems (such as Wikipedia, Flickr, YouTube, Facebook, MySpace) appeared, marking the arrival of Web 2.0. This Web is growing rapidly, with many new CS systems being developed and non-CS systems adding CS features.

The user interface should make it easy for users to contribute. This is highly non-trivial.

These CS systems often combine multiple basic CS features. For example, Wikipedia primarily builds a textual KB. But it also builds a structured KB (via infoboxes) and hosts many knowledge sharing forums (for example, discussion groups). YouTube lets users both share and evaluate videos. Community portals often combine all CS features discussed so far. Finally, we note that the Semantic Web, an ambitious attempt to add structure to the Web, can be viewed as a CS attempt to share structured data, and to integrate such data to build a Web-scale structured KB. The World-Wide Web itself is perhaps the largest CS system of all, encompassing everything we have discussed.

Challenges and Solutions

Here, we discuss the key challenges of CS systems:

How to recruit and retain users? Recruiting users is one of the most important CS challenges, for which five major solutions exist. First, we can *require users* to make contributions if we have the authority to do so (for example, a manager may require 100 employees to help build a company-wide system). Second, we can *pay users*. Mechanical Turk for example provides a way to pay users on the Web to help with a task. Third, we can *ask for volunteers*. This solution is free and easy to execute, and hence is most popular. Most current CS systems on the Web (such as Wikipedia, YouTube) use this solution. The downside of volunteering is that it is hard to predict how many users we can recruit for a particular application.

The fourth solution is *to make users pay for service*. The basic idea is to require the users of a system *A* to "pay" for using *A*, by contributing to a CS system *B*. Consider for example a blog website (that is, system *A*), where a user *U* can leave a comment only after solving a puzzle (called a captcha) to prove that *U* is a human. As a part of the puzzle, we

can ask *U* to retype a word that an OCR program has failed to recognize (the "payment"), thereby contributing to a CS effort on digitizing written text (that is, system *B*). This is the key idea behind the reCAPTCHA project.[34] The MOBS project[12,13] employs the same solution. In particular, it ran experiments where a user U can access a Web site (such as a class homepage) only after answering a relatively simple question (such as, is string "1960" in "born in 1960" a birth date?). MOBS leverages the answers to help build a data integration system. This solution works best when the "payment" is unintrusive or cognitively simple, to avoid deterring users from using system *A*.

The fifth solution is to *piggyback on the user traces* of a well-established system (such as building a spelling correction system by exploiting user traces of a search engine, as discussed previously). This gives us a steady stream of users. But we must still solve the difficult challenge of determining how the traces can be exploited for our purpose.

Once we have selected a recruitment strategy, we should consider how to further encourage and retain users. Many *encouragement and retention (E&R) schemes* exist. We briefly discuss the most popular ones. First, we can provide *instant gratification*, by immediately showing a user how his or her contribution makes a difference.[16] Second, we can provide an *enjoyable experience or a necessary service*, such as game playing (while making a contribution).[32] Third, we can provide ways to *establish, measure, and show fame/trust/reputation*.[7,13,24,25] Fourth, we can set up competitions, such as showing top rated users. Finally, we can provide *ownership situations*, where a user may feel he or she "owns" a part of the system, and thus is compelled to "cultivate" that part. For example, zillow.com displays houses and estimates their market prices. It provides a way for a house owner to claim his or her house and provide the correct data (such as number of bedroomss), which in turn helps improve the price estimation.

These E&R schemes apply naturally to volunteering, but can also work well for other recruitment solutions. For example, after *requiring* a set of users to contribute, we can still provide instant gratification, enjoyable experience, fame management, and so on, to maximize user participation. Finally, we note that deployed CS systems often employ a mixture of recruitment methods (such as bootstrapping with "requirement" or "paying," then switching to "volunteering" once the system is sufficiently "mature").

Given the success of current crowdsourcing systems, we expect that this emerging field will grow rapidly.

What contributions can users make? In many CS systems the kinds of contributions users can make are somewhat limited. For example, to evaluate, users review, rate, or tag; to share, users add items to a central Web site; to network, users link to other users; to find a missing boat in satellite images, users examine those images.

In more complex CS systems, however, users often can make a far wider range of contributions, from simple low-hanging fruit to cognitively complex ones. For example, when building a structured KB, users can add a URL, flag incorrect data, and supply attribute-value pairs (as low-hanging fruit).[3,5] But they can also supply inference rules, resolve controversial issues, and merge conflicting inputs (as cognitively complex contributions).[25] The challenge is to define this range of possible contributions (and design the system such that it can gather a critical crowd of such contributions).

Toward this goal, we should consider four important factors. First, how *cognitively demanding* are the contributions? A CS system often has a way to classify users into groups, such as guests, regulars, editors, admins, and "dictators." We should take care to design cognitively appropriate contribution types for different user groups. Low-ranking users (such as guests, regulars) often want to make only "easy" contributions (such as answering a simple question, editing one to two sentences, flagging an incorrect data piece). If the cognitive load is high, they may be reluctant to participate. High-ranking users (such as editors, admins) are more willing to make "hard" contributions (such as resolving controversial issues).

Second, what should be the *impact* of a contribution? We can measure the potential impact by considering how the contribution potentially affects the CS system. For example, editing a sentence in a Wikipedia page largely affects only that page, whereas revis-

ing an edit policy may potentially affect million of pages. As another example, when building a structured KB, flagging an incorrect data piece typically has less potential impact than supplying an inference rule, which may be used in many parts of the CS system. Quantifying the potential impact of a contribution type in a complex CS system may be difficult.[12,13] But it is important to do so, because we typically have far fewer high-ranking users such as editors and admins (than regulars, say). To maximize the total contribution of these few users, we should ask them to make potentially high-impact contributions whenever possible.

Third, what about *machine contributions*? If a CS system employs an algorithm for a task, then we want human users to make contributions that are easy for humans, but *difficult for machines*. For example, examining textual and image descriptions to decide if two products match is relatively easy for humans but very difficult for machines. In short, the CS work should be distributed between human users and machines according to what each of them is best at, in a complementary and synergistic fashion.

Finally, the user interface should make it easy for users to contribute. This is highly non-trivial. For example, how can users easily enter domain knowledge such as "no current living person was born before 1850" (which can be used in a KB to detect, say, incorrect birth dates)? A natural language format (such as in openmind.org) is easy for users, but difficult for machines to understand and use, and a formal language format has the reverse problem. As another example, when building a structured KB, contributing attribute-value pairs is relatively easy (as Wikipedia infoboxes and Freebase demonstrate). But contributing more complex structured data pieces can be quite difficult for naive users, as this often requires them to learn the KB schema, among others.[5]

How to combine user contributions? Many CS systems do not combine contributions, or do so in a loose fashion. For example, current evaluation systems do not combine reviews, and combine numeric ratings using relatively simple formulas. Networking systems simply link contributions (homepages and friendships) to form a social network graph. More complex CS systems, however, such as those that build software, KBs, systems, and games, combine contributions more tightly. Exactly how this happens is application dependent. Wikipedia, for example, lets users manually merge edits, while ESP does so automatically, by waiting until two users agree on a common word.

No matter how contributions are combined, a key problem is to decide what to do if users differ, such as when three users assert "A" and two users "not A." Both automatic and manual solutions have been developed for this problem. Current automatic solutions typically combine contributions weighted by some user scores. The work[12,13] for example lets users vote on the correctness of system components (the semantic mappings of a data integration systems in this case[20]), then combines the votes weighted by the trustworthiness of each user. The work[25] lets users contribute structured KB fragments, then combines them into a coherent probabilistic KB by computing the probabilities that each user is correct, then weighting contributed fragments by these probabilities.

Manual dispute management solutions typically let users fight and settle among themselves. Unresolved issues then percolate up the user hierarchy. Systems such as Wikipedia and Linux employ such methods. Automatic solutions are more efficient. But they work only for relatively simple forms of contributions (such as voting), or forms that are complex but amenable to algorithmic manipulation (such as structured KB fragments). Manual solutions are still the currently preferred way to combine "messy" conflicting contributions.

To further complicate the matter, sometimes not just human users, but machines also make contributions. Combining such contributions is difficult. To see why, suppose we employ a machine *M* to help create Wikipedia infoboxes.[35] Suppose on Day 1 *M* asserts population = 5500 in a city infobox. On Day 2, a user *U* may correct this into population = 7500, based on his or her knowledge. On Day 3, however, *M* may have managed to process more Web data, and obtained higher confidence that population = 5500 is indeed correct. Should *M* override *U*'s assertion? And if so, how can M explain its reasoning to *U*? The main problem here is it is difficult for a machine to enter into a manual dispute with a human user. The currently preferred method is for *M* to alert *U*, and then leave it up to *U* to decide what to do. But this method clearly will not scale with the number of conflicting contributions.

How to evaluate users and contributions? CS systems often must manage malicious users. To do so, we can use a combination of techniques that block, detect, and deter. First, we can block many malicious users by limiting who can make what kinds of contributions. Many e-science CS systems, for example, allow anyone to submit data, but only certain domain scientists to clean and merge this data into the central database.

Second, we can detect malicious users and contributions using a variety of techniques. Manual techniques include monitoring the system by the owners, distributing the monitoring workload among a set of trusted users, and enlisting ordinary users (such as flagging bad contributions on message boards). Automatic methods typically involve some tests. For example, a system can ask users questions for which it already knows the answers, then use the answers of the users to compute their reliability scores.[13,34] Many other schemes to compute users' reliability/trust/fame/reputation have been proposed.[9,26]

Finally, we can deter malicious users with threats of "punishment." A common punishment is banning. A newer, more controversial form of punishment is "public shaming," where a user *U* judged malicious is publicly branded as a malicious or "crazy" user for the rest of the community (possibly without *U*'s knowledge). For example, a chat room may allow users to rate other users. If the (hidden) score of a user *U* goes below a threshold, other users will only see a mechanically garbled version of *U*'s comments, whereas *U* continues to see his or her comments exactly as written.

No matter how well we manage malicious users, malicious contributions often still seep into the system. If so, the CS system must find a way to undo those. If the system does not combine

contributions (such as reviews) or does so only in a loose fashion (such as ratings), undoing is relatively easy. If the system combines contributions tightly, but keeps them localized, then we can still undo with relatively simple logging. For example, user edits in Wikipedia can be combined extensively within a single page, but kept localized to that page (not propagated to other pages). Consequently, we can undo with page-level logging, as Wikipedia does. Hoever, if the contributions are pushed deep into the system, then undoing can be very difficult. For example, suppose an inference rule *R* is contributed to a KB on Day 1. We then use *R* to infer many facts, apply other rules to these facts and other facts in the KB to infer more facts, let users edit the facts extensively, and so on. Then on Day 3, should R be found incorrect, it would be very difficult to remove *R* without reverting the KB to its state on Day 1, thereby losing all good contributions made between Day 1 and Day 3.

At the other end of the user spectrum, many CS systems also identify and leverage influential users, using both manual and automatic techniques. For example, productive users in Wikipedia can be recommended by other users, promoted, and given more responsibilities. As another example, certain users of social networks highly influence buy/sell decisions of other users. Consequently, some work has examined how to automatically identify these users, and leverage them in viral marketing within a user community.[24]

Conclusion

We have discussed CS systems on the World-Wide Web. Our discussion shows that crowdsourcing can be applied to a wide variety of problems, and that it raises numerous interesting technical and social challenges. Given the success of current CS systems, we expect that this emerging field will grow rapidly. In the near future, we foresee three major directions: more generic platforms, more applications and structure, and more users and complex contributions.

First, the various systems built in the past decade have clearly demonstrated the value of crowdsourcing. The race is now on to move beyond building individual systems, toward building general CS platforms that can be used to develop such systems quickly.

Second, we expect that crowdsourcing will be applied to ever more classes of applications. Many of these applications will be formal and structured in some sense, making it easier to employ automatic techniques and to coordinate them with human users.[37–40] In particular, a large chunk of the Web is about data and services. Consequently, we expect crowdsourcing to build structured databases and structured services (Web services with formalized input and output) will receive increasing attention.

Finally, we expect many techniques will be developed to engage an ever broader range of users in crowdsourcings, and to enable them, especially naïve users, to make increasingly complex contributions, such as creating software programs and building mashups (without writing any code), and specifying complex structured data pieces (without knowing any structured query languages). C

References

1. AAAI-08 Workshop. Wikipedia and artificial intelligence: An evolving synergy, 2008.
2. Adamic, L.A., Zhang, J., Bakshy, E. and Ackerman, M.S. Knowledge sharing and Yahoo answers: Everyone knows something. In *Proceedings of WWW*, 2008.
3. Chai, X., Vuong, B., Doan, A. and Naughton, J.F. Efficiently incorporating user feedback into information extraction and integration programs. In *Proceedings of SIGMOD*, 2009.
4. The Cimple/DBLife project; http://pages.cs.wisc.edu/~anhai/projects/cimple.
5. DeRose, P., Chai, X., Gao, B.J., Shen, W., Doan, A., Bohannon, P. and Zhu, X. Building community Wikipedias: A machine-human partnership approach. In *Proceedings of ICDE*, 2008.
6. Fuxman, A., Tsaparas, P., Achan, K. and Agrawal, R. Using the wisdom of the crowds for keyword generation. In *Proceedings of WWW*, 2008.
7. Golbeck, J. Computing and applying trust in Web-based social network, 2005. Ph.D. Dissertation, University of Maryland.
8. Ives, Z.G., Khandelwal, N., Kapur, A., and Cakir, M. Orchestra: Rapid, collaborative sharing of dynamic data. In *Proceedings of CIDR*, 2005.
9. Kasneci, G., Ramanath, M., Suchanek, M. and Weiku, G. The yago-naga approach to knowledge discovery. *SIGMOD Record 37*, 4, (2008), 41–47.
10. Koutrika, G., Bercovitz, B., Kaliszan, F., Liou, H. and Garcia-Molina, H. Courserank: A closed-community social system through the magnifying glass. In *The 3rd Int'l AAAI Conference on Weblogs and Social Media (ICWSM)*, 2009.
11. Little, G., Chilton, L.B., Miller, R.C. and Goldman, M. Turkit: Tools for iterative tasks on mechanical turk, 2009. Technical Report. Available from glittle.org.
12. McCann, R., Doan, A., Varadarajan, V., and Kramnik, A. Building data integration systems: A mass collaboration approach. In *WebDB*, 2003.
13. McCann, R., Shen, W. and Doan, A. Matching schemas in online communities: A Web 2.0 approach. In *Proceedings of ICDE*, 2008.
14. McDowell, L., Etzioni, O., Gribble, S.D., Halevy, A.Y., Levy, H.M., Pentney, W., Verma, D. and Vlasseva, S. Mangrove: Enticing ordinary people onto the semantic web via instant gratification. In *Proceedings of ISWC*, 2003.
15. Mihalcea, R. and Chklovski, T. Building sense tagged corpora with volunteer contributions over the Web. In *Proceedings of RANLP*, 2003.
16. Noy, N.F., Chugh, A. and Alani, H. The CKC challenge: Exploring tools for collaborative knowledge construction. *IEEE Intelligent Systems 23*, 1, (2008) 64–68.
17. Noy, N.F., Griffith, N. and Munsen, M.A. Collecting community-based mappings in an ontology repository. In *Proceedings of ISWC*, 2008.
18. Olson, M. The amateur search. *SIGMOD Record 37*, 2 (2008), 21–24.
19. Perkowitz, M. and Etzioni, O. Adaptive web sites. *Comm. ACM 43*, 8 (Aug. 2000).
20. Rahm, E. and Bernstein, P.A. A survey of approaches to automatic schema matching. *VLDB J. 10*, 4, (2001), 334–350.
21. Ramakrishnan, R. Collaboration and data mining, 2001. Keynote talk, KDD.
22. Ramakrishnan, R., Baptist, A., Ercegovac, A., Hanselman, M., Kabra, N., Marathe, A. and Shaft, U. Mass collaboration: A case study. In *Proceedings of IDEAS*, 2004.
23. Rheingold, H. *Smart Mobs.* Perseus Publishing, 2003.
24. Richardson, M. and Domingos, P. Mining knowledge-sharing sites for viral marketing. In *Proceedings of KDD*, 2002.
25. Richardson, M. and Domingos, P. Building large knowledge bases by mass collaboration. In *Proceedings of K-CAP*, 2003.
26. Sarwar, B.M., Karypis, G., Konstan, J.A. and Riedl, J. Item-based collaborative filtering recommendation algorithms. In *Proceedings of WWW*, 2001.
27. Steinmetz, R. and Wehrle, K. eds. Peer-to-peer systems and applications. *Lecture Notes in Computer Science. 3485*; Springer, 2005.
28. Stork, D.G. Using open data collection for intelligent software. *IEEE Computer 33*, 10, (2000), 104–106.
29. Surowiecki, J. *The Wisdom of Crowds.* Anchor Books, 2005.
30. Tapscott, D. and Williams, A.D. *Wikinomics.* Portfolio, 2006.
31. Time. Special Issue Person of the year: You, 2006; http://www.time.com/time/magazine/article/0,9171,1569514,00.html.
32. von Ahn, L. and Dabbish, L. Labeling images with a computer game. In *Proc. of CHI*, 2004.
33. von Ahn, L. and Dabbish, L. Designing games with a purpose. *Comm. ACM 51*, 8 (Aug. 2008), 58–67.
34. von Ahn, L., Maurer, B., McMillen, C., Abraham, D. and Blum, M. Recaptcha: Human-based character recognition via Web security measures. *Science 321*, 5895, (2008), 1465–1468.
35. Weld, D.S., Wu, F., Adar, E., Amershi, S., Fogarty, J., Hoffmann, R., Patel, K. and Skinner, M. Intelligence in Wikipedia. *AAAI*, 2008.
36. Workshop on collaborative construction, management and linking of structured knowledge (CK 2009), 2009. http://users.ecs.soton.ac.uk/gc3/iswc-workshop.
37. Franklin, M, Kossman, D., Kraska, T, Ramesh, S, and Xin, R. CrowdDB: Answering queries with crowdsourcing. In *Proceedings of SIGMOD 2011.*
38. Marcus, A., Wu, E. and Madden, S. Crowdsourcing databases: Query processing with people. In *Proceedings of CRDR 2011.*
39. Parameswaran, A., Sarma, A., Garcia-Molina, H., Polyzotis, N. and Widom, J. Human-assisted graph search: It's okay to ask questions. In *Proceedings of VLDB 2011.*
40. Parameswaran, A. and Polyzotis, N. Answering queries using humans, algorithms, and databases. In *Proceedings of CIDR 2011.*

AnHai Doan (anhai@cs.wisc.edu) is an associate professor of computer science at the University of Wisconsin-Madison and Chief Scientist at Kosmix Corp.

Raghu Ramakrishnan (ramakris@yahoo-inc.com) is Chief Scientist for Search & Cloud Computing, and a Fellow at Yahoo! Research, Silicon Valley, CA, where he heads the Community Systems group.

Alon Y. Halevy (halevy@google.com) heads the Structured Data Group at Google Research, Mountain View, CA.