ENGINEERING CROWDSOURCED STREAM PROCESSING SYSTEMS

MUHAMMAD IMRAN[1], IOANNA LYKOURENTZOU[2], YANNICK NAUDET[2], CARLOS CASTILLO[1]

[1] *Qatar Computing Research Institute,*
Doha, Qatar
mimran@qf.org.qa, chato@acm.org

[2] *CRP Henri Tudor*
Luxembourg
ioanna.lykourentzou@tudor.lu, yannick.naudet@tudor.lu

A crowdsourced stream processing system (CSP) is a system that incorporates crowdsourced tasks in the processing of a data stream. This can be seen as enabling crowdsourcing work to be applied on a sample of large-scale data at high speed, or equivalently, enabling stream processing to employ human intelligence. It also leads to a substantial expansion of the capabilities of data processing systems. Engineering a CSP system requires the combination of human and machine computation elements. From a general systems theory perspective, this means taking into account inherited as well as emerging properties from both these elements. In this paper, we position CSP systems within a broader taxonomy, outline a series of design principles and evaluation metrics, present an extensible framework for their design, and describe several design patterns. We showcase the capabilities of CSP systems by performing a case study that applies our proposed framework to the design and analysis of a real system (AIDR) that classifies social media messages during time-critical crisis events. Results show that compared to a pure stream processing system, AIDR can achieve a higher data classification accuracy, while compared to a pure crowdsourcing solution, the system makes better use of human workers by requiring much less manual work effort.

Keywords: Crowdsourcing, Stream Processing

1 Introduction

Stream processing refers to computations performed on an unbounded, high-speed, continuous, and time-varying sequence (stream) of events [49], which typically cannot be stored offline and needs to be processed on-line. If performed timely and effectively, stream processing can play a vital role in supporting real-time decision making. Many practical applications have been developed based on the stream processing paradigm, including real-time road condition monitoring, event-based business processing and profile-based query processing, implemented through various stream processing frameworks, such as the Aurora, TelegraphCQ, S4 and Storm project system among others [12, 13, 52, 60].

A significant drawback of traditional stream processing systems is that they rely entirely on automated algorithms of data processing, and as such they are limited by the processing capabilities of these algorithms. Indeed, it is often the case that data stream to be processed is imprecise, highly variable and previously unseen, yet the decision based on this data needs to be made in real-time. In this case, automated processing on its own may produce undesirable results, harming the decision-making process it needs to support. For example, suppose we are interested in monitoring social media to identify messages related to a major crisis event (e.g. an epidemic spreading such as the H1N1 flu pandemic in 2009). The moment this data is issued it needs to be correctly categorized and classified, in order to inform health and policy decisions accurately and on time. Unfortunately, relying only on automated data processing algorithms may not be adequate, firstly because user-generated data in Twitter may partially be erroneous, due to human mistake or intentionally, and secondly because new data may have different characteristics than existing data used to create the algorithms used by the system. In machine learning, this is referred to as the *domain adaptation problem*, where due to concept drift and other issues, a model trained on one event does not perform well on another. Back to our previous example of epidemic spread detection, a fully automatic stream processing algorithm trained on data from the H1N1 epidemic, may not function correctly on a new epidemic (e.g. the MERS-coronavirus outbreak in 2014), for instance due to the presence of unique vectors (e.g. spread through camels instead of birds) which have not been seen in previous situations. Nevertheless, failing to correctly filter the stream data of the new situation may lead to incorrect conclusions. It therefore becomes apparent that *in cases where critical—in terms of cost, time or other valuable resource—decision-making needs to take place in real-time, based on data streams that are potentially noisy and unseen, fully automated stream processing systems do not suffice.* Similar scenarios to the above include detecting intrusion or fraud based on anomalous or unusual activities.

Crowdsourcing is a recent computing paradigm that has been used to address the above problems [29]. It refers to utilizing a large number of users (a crowd) to perform a task, by splitting it into several sub-tasks and allocating them to individual workers. Crowdsourcing presents the significant advantage of involving human intelligence; and with it, the ability of humans to categorize, filter and evaluate, often better than automated methods. In fact, crowdsourcing has proven to be better than current computer algorithms in numerous cases, ranging from image recognition and audio transcription to real-time document syntax checking. In our example above, crowdsourcing could make use of the ability of humans to make better sense of a textual message than any algorithm, and in this sense help categorize and disambiguate the information included in a crisis data stream much more effectively. Given that typical crowdsourcing relies entirely on human effort, which inevitably entails the risk for low recognition quality, typical crowdsourcing approaches apply redundancy, i.e. they allocate the same task to multiple workers. This redundancy decreases the throughput and increases the cost of crowdsourced processing. Adding to this problem is the fact that stream processing tasks, unlike other types of input to crowdsourcing, necessitate a very high response rate (throughput) from the crowd. For example, during the Hurricane Sandy a peak rate of 16,000 tweets/minute was observed, and in such case, pure crowdsourcing may easily fail due to human limitations. *Summarizing, while fully automated processing may not be appropriate because of errors or quality issues, fully crowdsourced processing may not be appropriate because of processing time (throughput) and cost.*

The above problem leads to the nascent class of crowdsourced stream processing systems (CSPs), which combine automatic and human processing elements applied on time-sensitive, often critical information in the form of data streams. Certain instances of such systems have recently appeared, requiring different amounts of automated and human involvement, and involving different architectural and design patterns, of which we provide an overview in the next sections. Despite their appeal and necessity, however, no concrete framework exists currently to organize the characteristics, functionalities, architectural patterns and problem cases that CSP systems apply on. As a result, the design of CSPs today is ad-hoc and, at best, intuition-based. A systematic formalization and synthesis of these systems could significantly improve the design and engineering of CSP applications, leading to better designs and to performance improvements of current systems, and guiding the development of new ones. Given the added value and criticality of accurate, fast stream processing, and the abundance of applications that need it, one may easily understand why such systematization is essential. To the best of our knowledge, *this is the first work attempting to systematize the description and analysis of CSPs through a concrete functional, architectural and evaluation framework.*

Contribution and organization of this paper. First, we propose a generic framework for the design and analysis of crowdsourced stream processing systems. Our contribution begins in Section 2 by positioning CSPs into a broader taxonomy of systems. Then we describe design objectives, principles and quantifiable evaluation criteria (Section 3). In Section 4 we introduce a generic application design framework in terms of composable elements and communication channels. In Section 5 we present a series of design patterns to solve specific design problems. Section 6 shows a case study in which we design and analyze a CSP system using our framework. Finally, Section 7 outlines related work and Section 8 summarizes our main conclusions.

2 Characterizing crowdsourced stream processing systems

In this section we introduce the main characteristics of crowdsourced stream processing systems (CSPs). We start by providing a systemic view on CSPs, giving their elements, functions and properties. Then, we provide tangible examples of CSPs using a series of systems previously described in the literature.

2.1 The CSP system

In order to better understand what a CSP system is, we follow a relaxed systemic-modeling approach [3], which borrows explanatory elements from systems science and General Systems Theory [6]. According to this theory, *a system is a set of interacting elements* that has *emerging properties* which are richer than the sum of the properties of its parts.

A system can be examined from two main perspectives [18]: (i) a *behavioral* or *teleological* perspective, where the system's behavior and goals are examined, and (ii) a *structural* perspective, where the system's structure, architecture and operations are considered. These perspectives are supported by notions that include environment, objective, function, element, relation and interface [59], as illustrated by the system model of Figure 1, adapted from [59]. *Environment* is anything outside the system's boundaries. It influences the system as well as the system influences it. *Objective* is the system's finality at a given time, a notion that directly influences the system's structure and functionality. *Function* is the set of actions the system can execute in order to realize its objective. *Element* is a component of a system, which can itself be a system. Finally, *Interface* is the element through which the system establishes connections with its environment, each system's element having interfaces through which *Relations* with other elements are established. Environment, objective and function provide the *behavioral perspective* of the system, i.e. they denote the way the system acts and reacts. Element and interface provide the *structural perspective* of the system, i.e. they materialize the internal organization and architecture of the system.

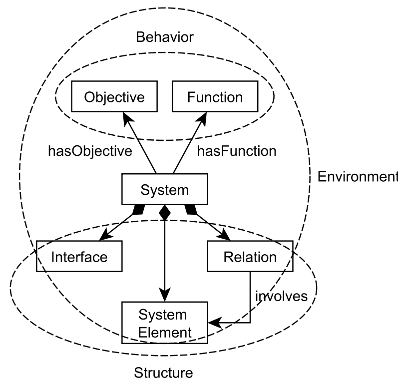


Fig. 1. Simplified meta model of a System. Diamond arrow represents the composition relation.

A CSP is a combination of two systems: a *Stream Processing system* and a *Crowdsourcing system*. It keeps the objective of the first, while using the other as a tool to better reach this objective.

Stream processing system. From a *behavioral perspective*, the objective of a stream processing system is to process data streams. Example applications include data mining [74], clustering [1, 85], classification [84], time series analysis [47, 86] and burst detection [75], among many others.

From a *structural perspective*, a stream processing system is a composition of *processing elements* that read an input data stream and write an output data stream. Typically, each processing element performs a fairly simple task and is thus easy to implement, debug and maintain. The data passed between elements are data streams as well. Apart from handling a particular type of data (data streams) and employing single-pass processing, another property of stream processing systems is that—at a structural level—each processing element is assumed to have limited memory/storage, i.e. the amount of data it can store is small in comparison to the total amount of data that passes through it [57]. Stream processing systems have important properties which will be evaluated to measure the system performances: *latency* (the time that it takes for a stream data item to be processed), *throughput* (the number of items processed/unit time), *load adaptability* (the response of the system to abrupt changes in the input load), *processing quality* (processing approximation error and probability of error, in case this system functions on the basis of an approximation algorithm).

Crowdsourcing system. From a *behavioral perspective*, the objective of a crowdsourcing system is to perform some tasks by using the intelligence of a large group of persons, called the crowd. Typically, the goal is to apply human intelligence to tasks that cannot be performed effectively by fully automated means. This is done by involving large numbers of workers who perform small portions of a larger task, improving the efficiency of more traditional ways of work organization by reducing completion time and cost.

From a *structural perspective*, a crowdsourcing system is composed of two subsystems interacting one with the other: an information system and the crowd. While the crowd brings along the power of its numbers and a large diversity of knowledge backgrounds, it still remains a human system that can have unpredictable behavior. Thus the crowd's influence on the crowdsourcing system is usually limited to the expected input of the former to the latter, an input that is moderated or filtered when necessary. When the crowd is given more freedom that can influence the functionality of the whole crowd system, more complex regulation mechanisms are put in place [58].

At a more detailed level, crowdsourcing systems vary significantly in their structure (for comprehensive surveys see [19, 83]). These variations depend on the application (knowledge or creativity-intensive or more mechanistic), type of input data (homogeneous or heterogeneous) and coordination mode (independent workers, collaboration, competition, mixed). Also, the type of task handled varies greatly from one crowdsourcing system to the other. Among the tasks that workers can execute, we may find: *providing input*, *correcting*, *validating* a given information, *comparing*, *categorizing*, but also *searching* for information, *synthesizing* knowledge and *providing judgments*, among others. This leads to prototypical classes of crowdsourcing tasks: binary classification, n-ary classification, and data entry contributions. In *binary classification*, workers are asked to give a "yes/no" answer to a given question, such as "do these suspicious activities constitute an attack?" [15], "does this entity correspond to this word in this context?" [16], "do these two records correspond to the same person?" [81]. Examples of *n-ary classification*, include determining if a word in a sentence represents a person, a place, or an organization [21], determining if an emergency-related tweet describes infrastructure damage, casualties, injuries, needs, etc. [38], determining which emotion corresponds to a tweet [49], performing relevance assessments [2], determining certain properties of a photo of a galaxy [42], etc. In *data-entry contributions*, workers are asked to perform a data-entry task such as: "rewrite this sentence in a shorter way" [4], "fill in the missing fields in this record" [23], "make a drawing of an articulated figure" [16], or provide free-text labels for an image [78]. Binary and n-ary classification tasks are usually more restricted, self-contained and can be more easily split to micro-task level, compared to data-entry ones.

Many real-time crowdsourcing applications (e.g. [23, 29]), are designed to have low latency, and to rely on homogeneous input data, highly decomposable and self-contained tasks, parallelizable task handling and centralized work coordination. This kind of crowdsourcing is referred to as *crowd processing* [29], and is the one that can be most easily integrated with stream processing systems to create CSPs.

As crowdsourcing is part of a CSP, key challenges of it are inherited by CSPs including dealing with the innate uncertainty of human work (e.g. workers might leave unexpectedly and their skills might vary [67]), and the fact that crowdsourcing is in general slower and more costly per item than fully automatic processing. Then, the crowdsourcing properties that will be usually of concern in the design of a CSP are the crowdsourcing *cost* (the cost of the work of the people providing the human intelligence) and the *quality* of the crowd input.

Crowdsourced stream processing system (CSP). CSPs lie in the junction between automated stream processing systems and crowd processing systems. They inherit behaviors and structural elements from both, while demonstrating its own emerging properties. What CSPs inherit from stream processing systems is their general objective: to process input

Table 1. Types of processing elements CSP systems contain, with relations and some indicative functions.

Type of Processing Elements	Functions
Automatic Processing Element (APE)	Computation Filtering Task generation Task assignment Task aggregation
Crowdsourced Processing Element (CSPE)	Binary classification N-ary classification Data entry

streams efficiently. What differentiates them is the *class of problems* that CSPs can handle. From crowdsourcing systems, CSPs inherit the capability of executing data processing tasks that require human intelligence, in contrast to the tasks traditional stream processing systems execute, which can be done by computer algorithms alone. In addition, CSP inherits the uncertainty/variability introduced by human work, and its costs, which necessitates methods and design choices to mitigate it.

From a *structural perspective* a CSP is composed of processing elements (PE), which involve either human or machine elements: crowdsourced PE (CSPE) and automatic PE (APE). These elements can be combined in different ways. We may distinguish two prototypical connection topologies, which generate three possible kind of relations:

- *Parallel.* In parallel connections, both automatic and crowdsourced processing elements function on the same type of task and layer of data stream processing, therefore such tasks have no dependencies.
- *Serial.* In serial connections, tasks have dependencies among them, requiring a certain degree of serialization in their processing.
- *Hybrid.* We remark that in general for solving real-world problems, hybrid topologies are used, involving a combination of parallel and serial connections.

In general, in a CSP the combination of automatic and crowdsourced processing elements is such that it *avoids having crowdsourced processing elements in high-volume critical paths*, employing them only in non-critical paths or in low-volume areas of a critical path. The volume of data to be processed by crowdsourced processing elements is often reduced by automatic processing elements performing filtering or aggregation operations.

From a *behavioral perspective* the objective of CSPs is to process data streams. This is supported by different functions given to the APE or CSPE elements, which we detail in the following. Some tasks that workers can execute within a CSP include *providing input* to other processing elements, *correcting* an element's output, *validating* that an element performed as expected, and *training* the automated processing elements so that they can learn to perform a task. The most suitable tasks for crowdsourced processing elements are those that involve *crowd processing* (such as binary and n-ary classification), while open-ended tasks are harder to integrate in a CSP because they may not be self-contained and are often non-decomposable.

Whereas crowdsourced processing elements deal with higher-difficulty tasks, automatic processing elements are employed for efficiency. Indicatively, we can distinguish between five

Table 2. Three indicative examples of CSPs.

Example CSPs	Crowdsourced processing	Automatic processing	Dominant composition
Network intrusion detection [63]	input; binary classification	filter	serial
Automatic online classification [81]	training; n-ary classification	task generation	hybrid
On-demand missing data completion [23]	input; comparison	task generation, aggregation	hybrid

functions for the APE (see Table 1). First, and perhaps most frequently, automatic elements can be used to perform arbitrary *computations* over the data (transform, aggregate, sort, join, etc. [56]). Secondly, automatic elements can be used for task *filtering*: performing sampling to reduce the number of elements that need to be processed. This may help to reduce the number of crowdsourcing calls by selectively using crowdsourcing only for difficult items (e.g. [16, 42]). Third, automatic elements can be used for automatic *task generation*, as in the case of systems that need to convert input streams into questions for humans (e.g. graph searching in [64], crowd-assisted data joins in [23]). Fourth, automatic elements can be used for *task assignment* purposes to ensure each crowdsourcing worker receives a task s/he is most capable of doing, e.g., with the online algorithm described in [32]. Finally, automatic elements can be used for *aggregation* of the output of other processing elements, for instance in order to ensure output quality through redundancy, as in the "get-another-label" system [41].

In section 4, we will provide a prototypical structure of a CSP and detail more precisely APE and CSPE and the way they function together.

2.2 Examples

We close this section by illustrating three indicative CSPs examples, categorized according to the functions of the automatic processing and crowdsourced processing elements and the kind of relation between them (Table 2).

Network intrusion detection. Automatic processing elements monitor network or system logs, to detect candidate malicious activities and generate alarm reports about possible attacks [63]. Human input is then needed to manually verify candidate alarms and take further action (perform remedy, re-tune the system) in response. Processing composition in this example is serial, with the automated element performing the filtering part (detect) and the human element performing binary classification (verified attack or not).

Automatic online classification. Crowdsourced processing elements are asked to correct n-ary classification of entities (e.g. mentions of people or places in a text) into semantic clusters, helping at the same time to train the automated element of the CSP into performing more accurate automatic classifications [81]. In this case, datasets can be static or dynamic (query-dependent). The processing composition in this system is hybrid (namely, a loop), with the automated element generating entity resolution questions, humans answering them and then these answers being sent back to the automated elements for training and decision purposes.

Table 3. Design objectives and principles

Design objective	Example metric	Design principles: Automatic components	Design principles: Crowdsourced components
Low Latency	End-to-end time	Keep-items-moving	Simple tasks
High Throughput	Output items per unit of time	High-performance processing	Task automation
Load Adaptability	Rate response function	Load shedding, load queueing	Task prioritization
Cost Effectiveness	Cost vs. quality/throughput/etc.	—	Task frugality
High Quality	Application-dependent	Redundancy/aggregation and quality control	

On-demand missing data completion. Crowdsourced processing elements are orchestrated to fill in missing data with respect to a query [23]. Among other operations, in this system crowdsourcing workers complete missing data in tuples that need to be selected or compared with other tuples in response to a query. Queries are not known beforehand, and the underlying dataset can change over time.

3 Design objectives, principles and metrics

In the previous section we presented a characterization of crowdsourced stream processing systems. In this section we give design principles that help building efficient systems. This efficiency can be achieved through optimising the important properties of CSPs inherited from its parents: stream processing and crowdsourced systems (see section 2.1). Each resulting design objective is quantifiable through a certain metric, and trade-offs among different objectives may occur, as summarized on Table 3.

The first three objectives (low latency, high throughput and load adaptability) are common in the stream processing literature (e.g. [56, 77]); while they are not crowdsourcing-specific, we adapt them to the crowdsourced stream processing setting. We do so by indicating how to address each objective in the automatic and crowdsourced components of the system. The last two objectives (cost effectiveness and high quality) are a consequence of the presence of humans as part of the system.

3.1 Low Latency

Latency is the time it takes for an item to be processed, and a low latency system takes less time to deliver results than a high latency system. In the literature, low-latency crowdsourcing stream processing systems have been referred to as real-time crowdsourcing systems (e.g. [5, 45]).

Measuring latency. A typical measure for the latency of a system is the average latency of the items traversing through it. The latency of an item is trivial to compute if the mapping from input to output items is one-to-one. If that is not the case (e.g. because each output item depends on several input items that arrived at different times), then for each input item that contributed to one or more output items, its latency can be measured as the time it took for the first such output item to be produced.

3.1.1 Automatic components: keep data moving

Automatic components should avoid unnecessary latencies due to network delays, storage operations, or other causes. Input items may be out-of-order, missing or delayed. Automatic processing elements should use non-blocking operations, never waiting indefinitely for some data items to arrive before continuing processing [73].

3.1.2 Crowdsourced components: simple tasks

Task design depends on many aspects that include incentives, interface, task description, and more importantly the task itself [22, 45]. Research and practice suggest to decompose complex tasks into simple subtasks, where each task is designed to be simple and follow specific requirements. This can not be understated, as it helps both to reduce the cost and to increase the quality of work. Simpler tasks are completed earlier [34], not only because each task is completed faster, but because the pool of people with the skills required to complete a task is larger if the task is simpler. A difficult task may introduce latency, and it may also reduce output quality [43].

Task decomposition helps reduce individual task complexity. The latency of the decomposed task may be lower if each individual task can be answered faster and/or by a potentially larger group of crowdsourcing workers. We note that task decomposition may have different constraints across applications. While in general tasks should be as simple as possible, in some cases it may be efficient to bundle a few sub-tasks together, e.g. to reduce context switching and reduce latency.

3.2 High Throughput

Throughput is the speed at which items are processed: a high-throughput system can process more items per unit of time than a low-throughput one. The throughput of a system is a function of its design and of the throughput of its components.

Measuring throughput. Throughput can be measured as output items per unit of time, e.g. items/second.

3.2.1 Automatic components: high performance

A stream processing system should process and accommodate long-running analysis requirements almost in real-time [77]. Each automatic processing element must be implemented to have high performance, and the application infrastructure including communication channels must be able to have a high throughput.

3.2.2 Crowdsourced components: task automation

In a crowdsourced stream processing system, in general the throughput of the crowdsourced processing components is lower than that of the crowdsourced processing components. Crowdsourced components may become a bottleneck.

Automating as much work as possible is a way of maintaining a high throughput: any aspect that can be automated should be not passed to a crowd. For instance in binary classification if we are searching for items belonging to a positive class, if an item can be automatically classified into the negative class, it does not need to be given to crowdsourcing workers.

3.3 Load Adaptability

The rate of input data may experience sudden changes including significant bursts. Adaptability is the capacity of the system to respond to a surge in demand.

Measuring load adaptability. Adaptability can be measured as the response function of throughput and latency vs input load. Ideally, these variables should not be strongly affected by increases in input load. In other words, we should not observe a significant reduction in throughput, or a significant increase in latency.

3.3.1 Automatic components: load shedding/queuing

Surges in input rate may require to use *load shedding*, this is, completely ignoring data items that are beyond the processing capabilities of a processing element at a given time [76]. Buffering may prevent shedding by allowing *load queuing*, using a bounded-size queue.

3.3.2 Crowdsourced components: task prioritization

The system should prevent crowdsourcing from becoming a bottleneck. This can be done by having a method for prioritizing tasks, in such a way that during periods of increased input rate, more tasks are taken by automatic parts of the system than normally. This may be done at the expense of output quality, if necessary.

For instance, Demartini et al. [16] pass to the crowd only tasks (in their case, entity linking tasks) for which an automatic system is uncertain, i.e. it has not given with high-confidence a positive or a negative answer. The thresholds of what constitute a high-confidence answer could be tunned to be able to handle more input items per unit of time.

3.4 Cost effectiveness

Given that crowdsourcing work is neither unlimited in supply nor free (even when done by volunteers, their time and motivation are precious resources), a principle of *task frugality* needs to be applied.

Crowdsourcing is usually compensated in proportion to the time spent by workers. Cost is therefore a function of (i) the payment per unit of time, (ii) of the number of items to process, (iii) the time needed by each worker to complete one item, and (iv) the plurality of workers per item. Effort is also a parameter, eventually translatable to cost as well. Increasing the payment per unit of time seems to have little effect in work quality, but can help reduce its latency [54].

The number of items to process is related to task automation—replacing crowd processing by automatic processing when possible—but also to the way processing elements are composed. Similarly to query planning in traditional database systems, heuristics or optimization methods can be used to determine the order in which automatic and crowdsourcing operations need to be done [23]. Task workflow design can be optimized through automatic processing elements, as described by Dai et al. [14].

The time and effort needed by each worker to complete one item can be decreased by decreasing task subjectivity and difficulty, for instance by decomposing into smaller subtasks, which may also lead to lower latency (Section 3.1.2). Task completion time can also be reduced by using more efficient worker-to-task allocation schemas, e.g., based on the average completion time and skills of each worker [9].

The plurality of workers per item reduces the effect of cheaters/spammers by using aggregated labels. This can be optimized by using a cost-sensitive objective, e.g., combining cost with quality as in [26].

Measuring cost effectiveness. This can be done in comparison with other aspects, e.g. cost/latency, cost/quality, etc. For instance, in [20] it is shown how larger budgets (in their case, fraction of items that are crowdsourced) yield better overall accuracy. The time to complete a task is also reduced when payment is increased [31, 54].

3.5 High quality

The addition of human elements makes crowdsourcing stream processing non-deterministic. Crowds vary in their composition and individual workers may provide varying levels of quality.

Output quality can be maintained by the interaction between automatic and crowdsourced components through operations such as aggregation, quality management and cheating detection (as mentioned e.g., in [83] and others). Section 5.1 describes a design pattern ("quality assurance loop") based on this type of interaction. Furthermore, the automated elements can play an important role in regards to incentives engineering, for example in diversifying task recommendations (bored workers perform worse than interested ones [44]), as well as in allocating incentives to tasks [82], with an aim to attract qualitative worker contributions and thus increase overall task quality.

Measuring quality. Quality should be measured in an application-dependent manner. The metric may be binary (correct item vs. incorrect item) or continuous, and it may not be a single scalar but comprise several aspects.

4 General CSP Framework

Taking into account the characteristics of CSP systems and the design principles presented above, in this section and the following one, we describe a framework for their design. In accordance with the CSP system model provided in section 2, this framework allows specifying the elements of a CSP system and communication flows among them, in a standardized way that allows to easily create various kinds of topologies, and to modify them when required. In the following we detail the proposed CSP design framework from a *system-level* (Section 4.1), *processing elements* (Section 4.2) and *communication flows* perspective. (Section 4.3).

4.1 System-level technical CSP meta-model

While the systemic model provided in section 2 offers a high-level view, Figure 2 of this section provides a technical meta-model for CSP systems. This figure follows the classical application-level description of software systems, with focus on stream processing systems (e.g. [27, 60]). According to this, the basic building blocks of stream processing systems are well-defined software components, which perform various kinds of jobs, and can be connected/composed together. Similarly, in the CSP meta-model we find the core elements of a CSP system: one or more processing elements *(PE)*, which can be computer-driven or human-driven (resp. *APE* or *CSPE*). Each PE performs dedicated tasks, and it may depend on other processing elements in terms of data requirements. The PEs are connected through communication flows for data or control, which are modeled using the *Data flow* and *Control flow* concepts. The data flow connections tie *input ports* and *output ports*. Data items are emitted from their

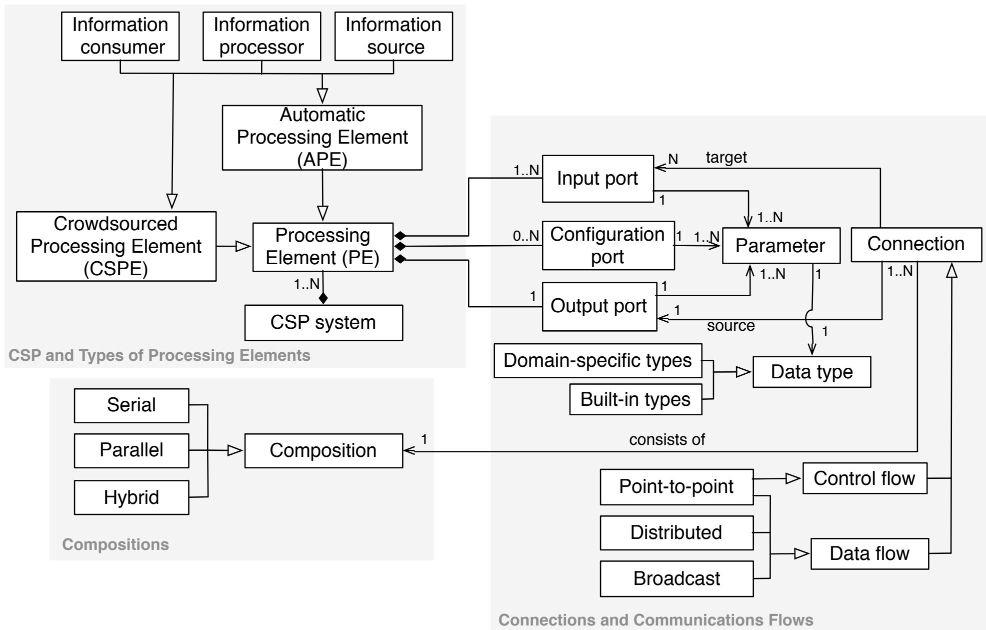


Fig. 2. Meta-model of a CSP system. The diagram depicts the model of processing elements, connections and communication flows between PEs and composition types.

source through a (usually single) output port and they are ingested by the target through (possibly multiple) input ports. Section 4.3 explains this asymmetry. Control connections can also be established to coordinate the behavior between pairs of PEs using (possibly multiple) configuration ports.

4.2 Processing elements

A processing element consumes data items and control flow signals through input ports, implements an application-specific requirement, and emits the processed data items through an output port. There are two types of processing elements as explained below.

4.2.1 Automatic Processing Elements

An automatic processing element (APE) is a standard component of stream processing systems, e.g. in [27, 60]. It executes a set of operations in a fully-automated manner on its input stream. Following the streaming computation model [57], we assume APEs do not have enough memory to hold all the items that go through them.

4.2.2 Crowdsourced Processing Elements

A crowdsourced processing element (CSPE) employs a large group of people (i.e. a crowd) to process data. The processing of one item in CSPEs is assumed to be more expensive and slow than in APEs. There are two ways in which this cost can be incorporated in the design of a system. A first option is to assume that the system can process at most k data items through CSPEs, i.e. there is a fixed *budget* for the crowdsourcing tasks that the system can do. A second option is to consider that there is a certain *cost per task* to be paid, and the system should be designed to minimize that cost. A CSPE may be implemented through the API of a local crowdsourcing platform (e.g. PyBossa)[a] or through the API of a remote crowdsourcing application (e.g. Amazon's Mechanical Turk,[b] CrowdFlower,[c] or others).

A processing element (PE) can act as information source, processor, or consumer as explained below.

Information source. Processing elements with no input ports are called *information source* elements, and perform *edge adaptation* [77], i.e. they convert external stimuli into data to be consumed by the CSP system. Information source PEs are responsible for emitting data continuously. An example of an automatic PE (APE) as information source is a sensor (e.g. seismometer that detects earthquakes) which gathers and provides continuous access to data. An example of a crowdsourced PE (CSPE) as an information source is the data generated through web-clicks of humans on the web.

Information processor. Processing elements with both input and output ports are called *information processors.*[d] An example of an APE as information processor, is an example of a computer program that performs some computation over data (e.g. a sort program). An example of a CSPE as information processor is the case of human-processed information (e.g. classification of a twitter message into positive or negative sentiments).

Information consumer. Processing elements with no output ports are called *information consumers*. Processing elements that only consume information—for example to preserve or visualize it—are known as information consumer elements. An example of an APE as information consumer, is a visualization component that ingests data only to visualize it. An example of a CSPE of type information consumer is a decision-maker (a human) that ingest data from computer program to make a decision during a crisis situation.

4.3 Connections and Communication flows

In this section, first, we describe the general communication behavior using channels and ports, second the communication modalities (i.e. *point-to-point*, *distributed* and *broadcasted*) and third the flows among PEs (i.e. *control flow* and *data flow*).

4.3.1 Channels and ports

Communication between processing elements is done through generic channels (sometimes referred to as *streams*). The concrete implementations of these channels may vary. In case buffering is required, the communication channels may be implemented using bounded-memory queues. If buffering is not required, other message-passing patterns can be used.

The processing elements have input, configuration, and output ports. The input ports are used to ingest data from different data sources. The configuration ports provide a way to configure a PE or to set default values for some of its parameters. The output port is used to emit the output data of the PE. Each port can have multiple parameters and the data types of those parameters can be set using either built-in types (primitive data types) or domain-specific types.

---

[a] `http://dev.pybossa.com/`
[b] `https://www.mturk.com/`
[c] `https://www.crowdflower.com/`
[d] In the Storm system [52] input sources are named "spouts", and information processors and consumers, "bolts".

4.3.2 Communication modalities

A single output port is all that is needed for most applications to emit output data. However, in addition to the one-to-one communication between two PEs, it is possible that output data has to be distributed from one source to many target PEs in parallel. To this end, the complexity of determining the destination PEs for each output item can be controlled by one of the following *communication modalities*:

Point-to-point. Point-to-point communications are the simplest case and have a single processing element as producer and a single processing element as consumer. Essentially, both source and target components in a point-to-point communication modality comprise one port each (i.e. source with the output port and target with the input port).

Distributed. The distribution communication modality supports cases where the output data of a processing element (e.g. data source PE) has to be ingested by multiple PEs. In such cases, multiple processing elements are subscribed to the same channel using the *distributed* communication model, possibly filtering data according to certain criteria or keys (as for example in [60]). The data items are distributed to multiple consumers.

Broadcasted. This communication modality is used when multiple processing elements need to process the same data items at the same time. The *broadcasting* communication model duplicates data items to all consumers that are subscribed to a channel.

4.3.3 Data and control flows

Data flows pass data items across processing elements. Data flows should use high-bandwidth channels with some amount of buffering, to provide better load adaptability.

Control flows allow a processing element to query or control the behavior of another processing element [8]. This can be used, for instance, to modify a parameter or a set of parameters in the operation of a processing element. Control flows should use low-latency channels with little or no buffering for faster response. Additionally, control flows should typically be used in point-to-point modality since the control signals may depend on the target processing element being addressed.

4.4 Composition types

A composition represents a specific topology used to connect various PEs in order to solve a specific problem. CSP systems can be composed using mainly three composition types, as follows.

Serial composition. A serial composition represents two or more PEs connected in a serial way, according to which the target PE always depends on the source PE in terms of data or control signal(s).

Parallel composition. Parallel composition represents the case where the connected PEs work in parallel and are independent of each other in terms of data or control signals.

Hybrid composition. Finally, the hybrid composition represents a mix of serial and parallel types and often a more complex composition of PEs.

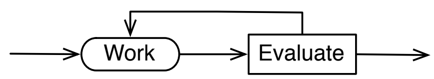
5 Design Patterns

The preceding section describes the generic technical components to implement when designing a CSP system. In this section we introduce specific design patterns for the connection of automated and crowdsourced processing elements. The idea of design patterns has been pivotal to software engineering since the work of Gamma et al. [25]. The design patterns presented in the section provide solutions to common problems with the CSPs context. *By its nature a list of design patterns is always open*, as new problems and new solutions to existing problems can be discovered and incorporated.

In the remainder of this paper, we make use of the following notation: rectangles represent automatic processing elements, rounded rectangles are crowdsourced processing elements, solid lines are data flows, and dashed lines are control flows.

5.1 Quality assurance loop

Schematic structure:



Problem. Given that the quality of work varies across workers, crowdsourcing is almost invariably used with some degree of redundancy. A fixed level of redundancy (e.g. "each task has to be done by 3 crowdsourced workers") may not be an optimal solution, given that some tasks may induce larger inter-worker agreement, thus requiring less redundancy, while other tasks may create larger disagreement and require more redundancy.

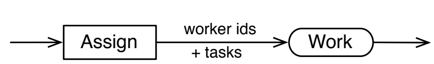
We would thus like to have per-task guarantees. In some cases, we would like to adaptively ask more workers to perform a task, until a certain criterion is satisfied. In other cases, we may want to ask for more expensive (but more accurate) workers to perform a certain task when we detect that regular workers are not agreeing enough. Some current crowdsourcing platforms offer this feature, by allowing to define the accuracy or expertise level of the involved workers.

Solution. Use an automatic quality assurance loop to aggregate and evaluate crowdsourcing work. The evaluation component may compute cost/quality trade-offs and stop when marginal accuracy increases are smaller than the cost of getting an extra label (as in [26]). The evaluation component may also keep a model of the trustworthiness of each individual worker, based on his/her past level of agreement (as in [41]).

Applicability. This pattern is applicable when the quality of the crowdsourcing work can be evaluated automatically by cross-checking labels from different workers. This pattern also applies when there is a global constraint that the output should honor, for example, if we are trying to obtain a total order on items that are being compared pairwise by workers. In this scenario, the evaluation component has to detect violations on the output (a violation of the transitivity property in this example), and ask for more crowdsourcing work to be done on the problematic subsets of the data.

5.2 User-specific task assignment

Schematic structure:



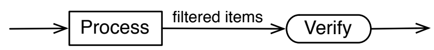
Problem. The crowd members are heterogeneous in their capacity to perform tasks. Assigning the "right" task to a worker leads to a reliable output, but failing to do so leads to an unreliable one.

Solution. An automatic processing element maintains a model of the skill of workers for different tasks. When feeding a task to a crowd, it indicates what is the specific worker-id that must perform each task. This design pattern should be combined with a quality assurance loop.

Applicability. This pattern is applicable when there is a variety of different skills needed for a given crowdsourcing job (comprising multiple tasks) and it is possible to automatically compute an estimation of the quality of the per-skill output of a worker performing a certain task (see for example [30] where workers of different sets of skills are matched to heterogeneous tasks through a mechanism design-based automatic element).

5.3 Process automatically, verify manually

Schematic structure:



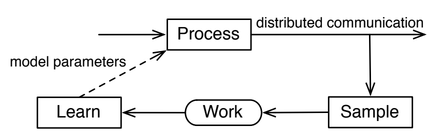
Problem. The input throughput exceeds the budget of crowdsourcing calls, so the data needs to be reduced before being crowdsourced.

Solution. An automatic processing element operating as a "detector" can act as a filter, and passes only the elements that are over a certain criterion towards a "verifier" crowdsourced processing element. For instance, crowdsourced content moderation (for profanity or hate speech or inappropriate content as in [55]) could pass only suspicious messages, containing certain keywords or image features, to a crowd of workers.

Applicability. In the context of binary classification tasks (other cases can be dealt with in a similar manner), this pattern is useful when there are methods to filter-out true negatives deterministically or with high precision. In this case, the number of crowdsourcing calls can be reduced by passing to the crowd only the examples that have a sufficient probability of being positive.

5.4 Supervised learning

Schematic structure:



Problem. The input throughput vastly exceeds the budget of crowdsourcing calls, so an automatic system needs to learn to mimic the work process performed by crowdsourced workers.

Solution. An automatic processing element runs a parametrized machine-learned model. Its output is sampled according to a certain criterion, and sent to a crowdsourced processing element to provide *training labels.* These labels are used to learn a new model, which is sent through a control signal to the main automatic processing element. The sampling can be done uniformly at random, or following the idea of *active learning* to maximize the gains in accuracy for every extra label. An example of this solution is Kamar et al. [42], who use machine vision to identify galaxies based on models learned from human labels.

Applicability. This pattern can be applied when it is possible to learn an automatic model of the process the crowd performs.

5.5 Crowdwork sub-task chaining

Schematic structure:

Problem. A complex task generates high latency and low quality output, and needs to be divided into separate parts.

Solution. Two or more crowdsourced processing elements can be composed on a serial, parallel, or hybrid circuit. The output from one CSPE is post-processed and sent to another CSPE (that can also be post-processed automatically). This is related to the now-centenarian paradigm of scientific management/Taylorism/Fordism. One example can be the decomposition of the task of counting calories in pictures of meals described in [61], where a large task (identifying nutrients in a certain food plate) is decomposed to several smaller sub-tasks (drawing boxes to identify the distinct food parts in a plate's image, tag and describe the identified parts, and measure their size in terms of portions). These sub-tasks are handled in a complex workflow within the original crowdsourced task, by different CSPEs, while additional PEs exist to help the process (for example calculating the level of agreement among workers). Similar examples where a large crowdsourcing task is divided into smaller sub-tasks and handled by separate CSPEs include the detect-fix-verify paradigm in [4], or a secondary grading task to control the quality of a primary task [72].

Applicability. This pattern can be applied when the task can be divided in advance into discrete sub-units. If not, this pattern can be extended by following an approach similar to [46], where workers are involved not only in solving the sub-tasks, but actually designing the workflow process to be followed, by suggesting decompositions of the original task. In this case, control flows need to connect the different crowdsourced processing elements, indicating changes in the tasks (e.g. decomposition) performed by them.

5.6 Human processing optional per item

Problem. A minimum throughput or maximum latency needs to be guaranteed, but crowdsourced stream processing elements can not guarantee that same throughput. Humans may become a bottleneck.

Solution. There should be a path of data flows between the application's input and its output that does not pass through any crowdsourced stream processing element. This means patterns such as *supervised learning* or other types of parallel connections need to be part of the design.

Applicability. This pattern can be applied when the minimum number of crowdsourcing calls needed in steady state for the application to function is zero. It can be applied after pattern presented in 5.4 when an adequate amount of training data has been gathered.

5.7 Human processing mandatory per item

Problem. A guaranteed level of quality, which can only be attained through human oversight, needs to be attained.

Solution. Every path of data flows between the application's input and output needs to pass through a crowdsourced stream processing element. One example is the case of a system to deal with reports of disruptive behavior (e.g. in a massive multi-player online game or similar system), where account suspension or other penalties can only be authorized by humans [33]. Additionally, a minimum level of redundancy can be applied in all such elements if multiple paths exist.

Applicability. This pattern can be applied when low latency does not need to be guaranteed. Enforcing that every data element that will be written in the output needs to pass through a crowdsourced component may introduce latency in the system.

6 Case Study: Classification of Social Media Messages during a Crisis

In this section, we study a concrete stream data processing problem, showing that neither pure crowdsourcing nor pure stream processing are satisfactory alternatives. Next, we show how the framework we have described can guide the CSP system design and serve as an analytical tool for its evaluation. Naturally, this real-world example does not use each and every one of the elements we have introduced, but it touches several aspects that are common to a variety of cases.

6.1 Problem Statement

In times of crises caused by natural hazards (such as floods and earthquakes), or by human intervention (such as civil unrest or war), people are increasingly turning to social media platforms such as Twitter.[e] In these platforms, users share information related to what they experience, what they need, what they witness, and/or what they consider important to repeat from other sources such as radio or television [35]. Social media usage "rises during disasters as people seek immediate and in-depth information" [24].

Several types of user are interested in this information [36], and their needs are varied. Members of affected communities need to take decisions about how to best protect their lives and property. Government and non-governmental response agencies need as much information as they can get in order to increase the effectiveness of their efforts. Different emergency response agencies are interested in different types of messages during different phases of an emergency [68]. For instance, reports of damage to infrastructures should be directed to some agencies, while reports about shortages of water and/or food should be directed to others.[f]

In this case study, we use Twitter platform to perform real-time acquisition and classification of users generated content posted online during crisis situations.

6.2 Crowdsourcing solution

Manual classification of messages is not possible given the scale of information that flows on Twitter, even under very generous assumptions.

To estimate the speed with which volunteers can solve these tasks, we use data obtained by Olteanu et al. [62] during the labeling of six crisis situations using paid crowdsourcing workers. The simplest task was "Indicate if tweets are informative for decision makers and emergency responders," with several classification options including "negative consequences," "donations

[e] `http://twitter.com/`
[f] The United Nations organizes its agencies into clusters: `http://business.un.org/en/documents/6852`.

or volunteering," "advice, warnings and/or preparation," among others (6 options in total). The crowdsourcing workers are divided into two groups: *trusted* and *untrusted*, depending on whether their judgments agree with that of the authors in a set of test questions (known as golden data). The average time per label for trusted workers was between 7 and 13 seconds, with an average of 9 seconds. Trusted workers were not allowed to label more than 250 items.

This means that one worker can produce 400 labels per hour, assuming s/he can keep that efficiency for such a long period. If we require 3 labels per item in order to have at least some degree of worker redundancy (recommended in a crowdsourcing setting to ensure quality), then a crowd of 3,000 volunteers will be needed to label 400,000 items per hour.

In a recent experience with digital volunteers during a large Typhoon in the Philippines in November 2013, the peak number of volunteers was 500, recruited through a number of channels including a large digital humanitarian network.[g] The peak throughput we can expect from such a crowd is in the order of 67,000 items per hour, or 1,100 tweets per minute. However, the largest documented peak of tweets per minute during a natural hazard that we are aware of is 16,000 tweets per minute.[h] *With pure crowdsourcing, we may be a factor of 10 slower than needed.*

6.3 Streaming solution

The automatic approaches to process data streams are indispensable yet challenging to adapt. The infinite length and evolving nature of data streams introduces concept-drift and concept-evolution issues, also known as the *domain adaptation problem* (an automatic machine-learned classifier trained with data from one situation does not perform well in another situation, as also discussed in Section 1). In order to incorporate the underlying changes which have occurred in the streams, incremental updating of the classification model is required by labeling new items, incorporating them as new training data, and learning a fresh model.

In particular for the crisis domain, although crises have some elements in common, they also comprise distinct elements which make domain adaptation difficult, and thus the automatic classification using pre-existing training data is not a satisfactory solution. For example, in [40], authors performed experiments on different crisis datasets, namely the Joplin tornado, the Sandy hurricane, and the Oklahoma tornado, which all struck different areas of the US in 2011, 2012 and 2013 respectively. The performance (AUC)[i] of different transfer scenarios was 0.52 (train on Joplin, test on Sandy), 0.56 (train on Joplin, test on Oklahoma) and 0.53 (train on Sandy, test on Oklahoma). Similar problems have been identified in another work [37]. This findings suggest that crisis-specific training data are needed since they lead to higher accuracy compared to training data from past disasters. *With pure automated stream processing, the quality achieved may be significantly low due to the domain adaptation problem.*

6.4 A Crowdsourced Stream Application: AIDR

The purpose of AIDR (Artificial Intelligence for Disaster Response),[j] is to filter and classify in real time messages posted in social media during humanitarian crises, such as natural

[g] `http://irevolution.net/2013/11/13/early-results-micromappers-yolanda/`
[h] During Hurricane Sandy in 2012: `http://www.cbsnews.com/8301-205_162-57542474/social-media-a-news-source-and-tool-during-superstorm-sandy/`
[i] AUC: area under the receiver operator characteristic (ROC) curve, a metric used for binary classification. AUC is measured in the [0,1] scale. The AUC score of a perfect classifier is equal to 1.
[j] `http://aidr.qcri.org/`

or man-made disasters [39]. Specifically, AIDR collects crisis-related messages from Twitter[k] ("tweets"), asks a crowd to label a sub-set of those messages, and learns an automatic classifier based on those labels. It also improves the classifier as more labels become available.

AIDR users begin by creating a collection process by entering a set of keywords that will be used to filter the Twitter stream. Next, they define the ontologies to be used to classify messages, selecting them from a set of pre-existing ones, or creating them from scratch. AIDR then instantiates an ongoing crowdsourcing task to collect training labels, which are used to train an automatic classifier that runs over the input data. Finally, an output of messages sorted into categories is generated, which can be collected and used to create crisis maps and other types of reports or in real-time decision-making.

6.4.1 Design overview

The design of AIDR follows the meta-model described in Section 4. At a high-level, the application uses the design pattern "Human processing", optional per item (Section 5.6), according to which items (Twitter messages) are able to traverse the application independently of the availability of crowdsourcing workers.

The processing elements of the application are composed following the diagram in Figure 3. The automatic processing elements include a Twitter collector, feature extractor, task generator, learner and classifier. A crowdsourcing processing element, the annotator, gathers human-provided labels. The *collector* is an information source automatic processing element that performs edge adaptation [77] to consume tweets using the Twitter streaming API. The *feature extractor* is an information process automatic processing element that prepares the messages by converting them to a set of textual features using standard text operations (extraction of unigrams, bigrams, part of speech classes, etc.).

The classifier, task generator, annotator and learner processing elements *interact* following the "supervised learning" design pattern (Section 5.4). The task generator samples the input stream, optionally performing de-duplication (to diversify the elements to label) and active learning (selecting elements for which the classification confidence with the current model is low). Both operations are implemented in a stream-aware way, in which the search for near-duplicates and low-confidence elements is done on a bounded-size buffer containing only the latest tweets consumed by the system. The learner keeps 20% of the labels it receives as a test set, and uses the remaining 80% for training, allowing it to report to the user the quality of the current classifier. The learner creates a new classification model (random forest in this case) every 50 training labels and transfers it to the classifier processing element using control signals.

The composition of the processing elements is done through publish/subscribe channels and queues following the hybrid composition model. The channels are able to broadcast and distribute data items, as well as to perform *load shedding* - discard elements in order to maintain throughput. The queues are capable of buffering data items, when necessary.

Implementation. The design of AIDR follows the *Service-Oriented Architecture* software architecture design pattern. All modules depicted in Figure 3 provide necessary RESTFul services enabling user interactions for the management purposes such as configurations, providing input, meta-data exchange. Moreover, the communication (in terms of real data)

[k] `http://twitter.com/`

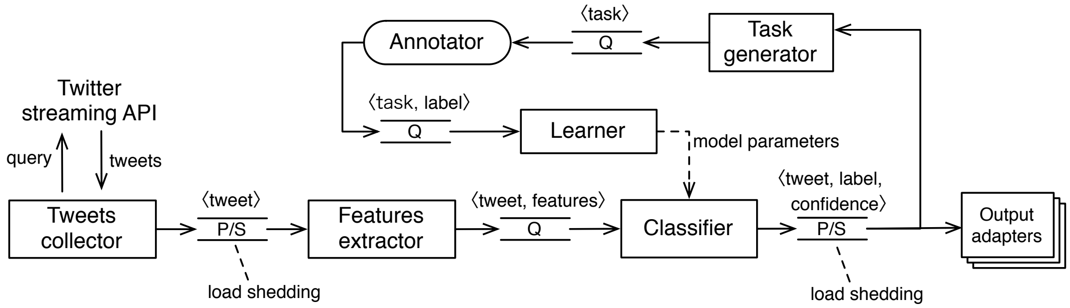


Fig. 3. AIDR general architecture. `P/S` indicates a publish/subscribe channel, while `Q` indicates a communication queue.

between modules happens using publish/subscribe channels and queues. REDIS[l] is used as an underlying communication infrastructure that supports pub/sub channels and queues.

The Java programming language and the Springs 3.0 framework is used for the main application logic, and ExtJS framework [m] for the application user-interfaces. PyBossa [n] is used as the crowdsourcing platform. The task generator modules passes newly selected items to be labeled to the PyBossa in order to employ crowdsourcing workers to label items.

6.4.2 Evaluation

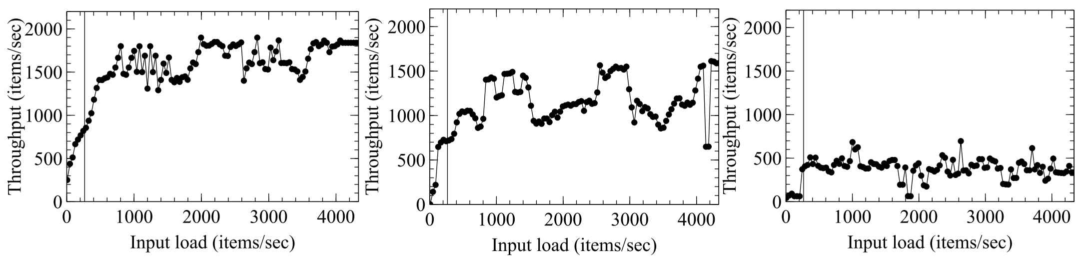
We apply the evaluation metrics described in Section 3. We remark that there are application-specific evaluation criteria that are outside the scope of our generic framework, e.g., user satisfaction in the case of interactive applications. Here we are interested in the performance (e.g. in high throughput and low latency) and quality (e.g. high classification accuracy of tweets) of using the AIDR CSP system.

We simulate the operation of AIDR with a fixed input stream under a varying set of streaming conditions. We note that we use real data in our simulation and mimic different streaming conditions to test the system's performance. For the purposes of this simulation, the application is instrumented to provide detailed information about data passing through it. *Mock objects* are also created to simulate data input (from Twitter's API) and data labeling (from a crowd).

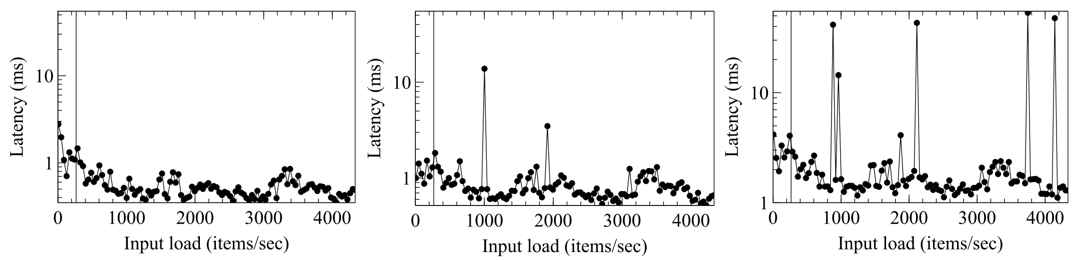
We use a dataset of 206,764 tweets containing the hashtag `#Joplin` and posted during the tornado that struck Joplin, Missouri in 2011 [37]. We also use 4,000 human-provided labels obtained via CrowdFlower. The specific crowdsourcing task was to indicate if a message is *informative* with respect to the disaster and of interest to a broad audience, or if it is either entirely of a personal nature or irrelevant for the disaster (the two latter classes are merged into a single class *not informative*).

Throughput, latency and load adaptability. We first measure the attributes that AIDR inherits from being a stream processing application. To evaluate these variables, its classifier is trained using all the available labeled data, and then varying input loads are applied.

---
[l] `http://redis.io/`
[m] `http://www.sencha.com/products/extjs/`
[n] `http://pybossa.com/`



(a) Throughputs: features extractor, classifier, system.



(b) Latencies: features extractor, classifier, system.

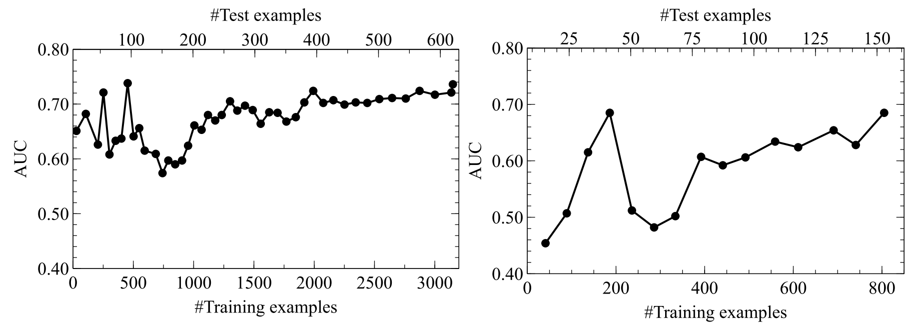
Fig. 4. Response of AIDR to varying input loads, in terms of throughput and latency. The vertical line corresponds to the highest peak rate documented for a natural disaster to date: 270 items/second during Hurricane Sandy in late 2012.

Figure 4 shows the results. In addition to end-to-end throughput and latency, we include a breakdown for each of the two main automatic processing elements of the system: the feature extractor and the classifier.

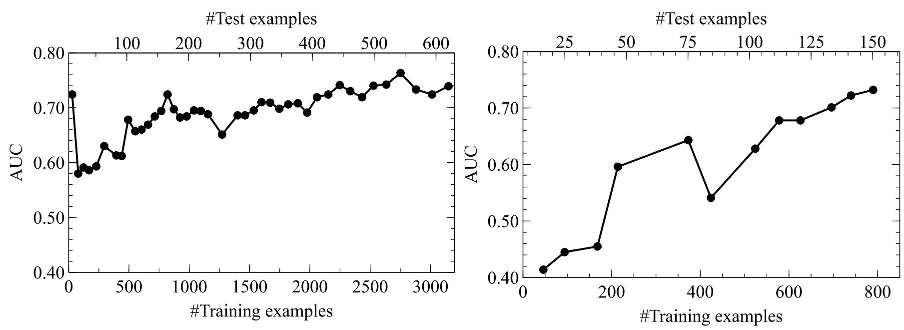
The system is designed following the principles of *keep data moving*, *high performance*, *task automation*, and *load shedding*, outlined in Section 3. The results indicate that the system is able to maintain a high throughput (500 items/second or more) above the observed peak rates in real disasters ($\approx$270 items/second). Because some of the items are dropped, the latency is kept in the order of tens of milliseconds, even when the input load exceeds the maximum output throughput. For the purposes of this application, this is acceptable given the large amount of redundancy in Twitter messages—about 1/3 are *re-tweets* in this dataset, and about half of the remainder are near-duplicates of another message.

Quality and cost. We next measure the attributes that AIDR inherits from being a crowdsourcing application, specifically, the relationship between quality and cost. Quality is measured using the area under the ROC curve (AUC); higher values are best and 0.5 indicates a random classifier. Cost is directly mapped to the number of human labels used.

Twitter data contains a significant amount of duplicates (i.e. re-tweets). Removing duplicates could decrease the crowdsourcing efforts (i.e. cost) needed to achieve an acceptable level of accuracy. To check this, we are interested in testing two variants of the AIDR system: i) how much cost (i.e. in terms of labeling effort) AIDR requires to achieve an acceptable quality using the passing learning approach and ii) the same using active learning approach, both with and without employing the de-duplication process (i.e. to discard duplicate tweets). In total, we examine four configurations of the task generator: (1) passive learning, equivalent to



(a) Quality vs. number of labels, using passive learning.



(b) Quality vs. number of labels, using active learning.

Fig. 5. Area under the ROC curve (AUC) shows classification quality vs. number of training labels shows cost (i.e. monetary in case of paid-workers, time in case of volunteers). The plots on the left side are obtained without de-duplication, while the plots on the right include it.

uniform random sampling in this case, (2) passive learning removing near-duplicate elements, (3) active learning, and (4) active learning removing near-duplicate elements. Results are shown in Figure 5 where we plot output quality vs. number of labels used by the classifier for training.

The system is designed following the principles of *simple tasks* and *task prioritization.* The test shows that even under conditions of *task frugality* the output quality is acceptable. After enough training data has been collected (≈1,000 labels), the AUC fluctuations stabilize and the system performs at an AUC of above 0.70. This point is reached with about half the labels if de-duplication is done and it collectively requires 1 minute of 500 workers, if they work simultaneously (as reported in Section 6.2). Further labels continue to increase the quality of the classifier, with diminishing returns. Active learning as implemented in this setting (over a bounded-size buffer) does not seem to yield significant improvements.

We remark that in Figure 5 every point represents a different (growing) testing set, a consequence of the online nature of the process. An offline analysis, where the testing set is fixed to 1/3 of the labeled elements, is in general consistent with the evaluation in the online setting. In the best case and using de-duplication, we obtain a maximum AUC of 0.64 for passive learning (after ≈270 labels) and an AUC of 0.66 (after ≈200 labels) for active learning. The offline analysis also shows that more labeled items further improve the results:

after using 2,600 labels we reach an AUC of 0.76, which is approximately equal to 3 minutes effort of 500 workers (as reported in Section 6.2).

Thus, *compared to a pure stream processing system, the AIDR CSP can be expected to achieve much higher quality* (AUC of 0.76, higher than pure SPs as explained in Section 6.3.)

On the other hand, using a pure crowdsourcing solution, one can achieve AUC 1.00 (by definition, because we are measuring quality against their labels) using a certain level of agreement among workers; however, to achieve such a high accuracy in a pure-stream processing system, we would need 500 workers continuously working for the duration of the event (which may span several hours or days). This is not feasible due to worker burn-out (performance deterioration, lower quality over time) and drop-out (less people over time). Instead, in a CSP system, to maintain high accuracy, we would only need these 500 workers for 3 minutes and then to re-use them for the same period every few hours–just to update the system (i.e. the supervised classification model in this case) with new training examples. Thus, *compared to a pure crowdsourcing system, the AIDR CSP achieves a high quality demanding much less effort by the crowd workers, thus using the available worker population more efficiently.*

In summary, this design enables human intelligence to be applied on a data intensive application. As a stream processing system, the presented CSP is able to keep up with the input loads of even large-scale disasters; and as a crowdsourcing application, it is able to use crowdsourcing work in an effective manner.

7 Related Work

Stream processing and crowdsourcing are vast research areas, so we focus on connecting our research to previous works covering topics closely related to ours. This includes engineering principles, frameworks, and taxonomies for the predecessors of CSPs, as well as example systems describing key design choices and best practices.

7.1 Data stream processing

Stream processing research covers many disparate fields (e.g., stock market data analysis [17], fraud detection system [71], intrusion detection systems [15], disaster prediction systems [11]), and has passed through a number of stages. Real-time stream processing systems perform data mining [74], clustering [1, 85], classification, time series analysis [47], and other decision support tasks [86]. Furthermore, to support the development of application-specific stream processing systems, there are general-purpose platforms such as S4 [60] and STORM [52] supporting scalable real-time processing. These systems are designed for high-speed continuous data ingestion, uninterrupted long-running processing, high-throughput, and low-latency.

We have incorporated key performance indicators from the above works into our evaluation metrics (Section 3), while our application design framework has been also inspired to some extent by the design of existing general-purpose stream platforms (Section 4).

7.2 Crowdsourcing

Two extensive crowdsourcing surveys can be found in [19, 83], which examine a variety of crowdsourcing systems – often under different names such as collective intelligence, human computation, or social systems.

7.2.1 Crowdsourcing system taxonomies

First, a number of works categorize crowdsourcing systems from an organizational theory point of view, classifying them according to their *business model functionality*. Geerts [28] performs a model-driven categorization, distinguishing the models of crowdcasting, crowdstorming, crowd production and crowdfunding. Vukovic [79] categorizes crowdsourcing systems by business-driven objectives, and further distinguish them in regards to their coordination model (marketplace or competitive-based). Lykourentzou et al. [50] distinguishes different user interaction modalities in crowdsourcing: collaborative, competitive, and hybrid. Saxton et al. [69] identify nine basic types which span from social financing to citizen media production models; one of their main conclusions is the need for crowd management which as we describe can be provided by a combination of processing elements, through certain design patterns.

Regarding *type-based classification*, crowdsourcing systems are classified per type of task (simple, complex, creative) in [70], while the different functions of the crowd (crowd rating, creation, processing and solving) are described in [29]. The first work covers "what" crowdsourcing workers do, and the second one covers "how" they do it, which is in line with the description of the role of humans in CSPs (Section 2).

There are other works that do not explicitly aim at a taxonomical ordering of crowdsourcing, but contain taxonomical and design elements. Quinn and Bederson [65] describe 3 dimensions related to our work: quality control, aggregation, and process order. The first is a design principle (Section 3.5). The second is an automatic element role (Section 2). The third describes interaction models between "computers, workers and requesters", and as such it is related to the design patters that we propose (Section 5).

7.2.2 Crowdsourcing frameworks and modeling approaches

Certain studies propose frameworks and modeling approaches to improve the design of crowdsourcing systems. Bozzon et al. [10] introduce a reactive crowdsourcing modeling approach focusing on the dynamic control of the crowd, by transforming high-level specifications (e.g. regarding task planning or worker handling) into elementary task-type executions. Roy et al. [67] propose SmartCrowd, a framework to interactively optimize three crowdsourcing processes: task generation, worker-to-task assignments and task evaluation, by taking into account the uncertainties introduced by the human factor. A number of works provide goal-driven design principles, which focus on the optimization of crowdsourcing systems for a global-level performance target. In this line, Lykourentzou et al. [51] present a stepwise modeling approach for the design of corporate crowdsourcing systems, realized in five decision-making and application steps (define goals, characterize jobs, profile workers, identify constraints and design a crowdsourcing optimization algorithm). Finally, Boutsis and Kalogeraki [9] focus on the process of task-to-worker allocation, and present a framework comprising four components (task management, dynamic assignment, profiling and scheduling), which aims at guaranteeing efficient crowdsourcing system performance under dynamic conditions of the crowdsourcing environment.

These works basically describe methodologies to optimize specific objectives of crowdsourcing system design. In contrast, our work describes a broad set of objectives and principles in a top-down manner (Section 3), providing a framework against which specific design methodologies can be implemented by composing different elements (Section 4).

7.3 Use Cases for CSPs

To the best of our knowledge, no prior work attempts to provide a general framework for engineering CSPs. However, there are works that describe CSP applications (without that specific name). We have already mentioned a number of these works in Section 2. Concrete examples include web table matching [20], entity resolution [81], or iterative text recognition [48]. A common element in them are high-throughput and low-latency requirements, which we capture in Section 3. There are also works describing real-time crowd-involving systems ("flash crowds" [45]), such as the system by Mashhadi and Capra [53] on quality control for user-contributed data in ubiquitous applications, the ones proposed by Bernstein et al. [4] and Bigham et al. [7], or the crisis response system described by Rogstadius et al. [66] (the latter has been recently adapted to operate with AIDR, the application we describe as a use case in Section 6).

An indicative example is CDAS [49], a CSP system for data analytics applied to tasks of sentiment analysis and image tagging. CDAS includes a quality assurance mechanism that takes into account the performance deviations of crowdsourced processing by monitoring historical performance data to estimate each elements' accuracy (instantiation of the quality evaluation metric of Section 3.5), as well as an online strategy to reduce waiting time (latency evaluation metric in our framework). Finally, systems using a mix of crowdsourced and automatic elements to achieve more efficient treatment of homogeneous data have been described, with two key examples described in [23, 80].

Certain elements of our framework borrow notions from the success and limitations showcased in the above systems. As an example, the study of performance in terms of quality, cost and speed, mentioned in the above works, have been expanded into the evaluation metrics (Section 3). Furthermore, several of the best practices illustrated above have been converted into design patterns (Section 5).

8 Conclusions

Crowdsourced stream processing is a new computational frontier that combines high-speed processing with human intelligence. Its progress hinges upon the development of efficient algorithms, as well as on the design of software architectures that implement those algorithms into applications having impact on the real world. This paper introduces a generic framework that covers system-level properties and behavior, design principles, structural elements, compositions and patterns, to enable better designs for future CSP applications and to serve as a basis for the evaluation and re-engineering of existing ones.

Much remains to be done between the design of specific CSP applications solving concrete problems, and the development of general frameworks and best practices that can serve as a basis for those designs. This includes extending specialized taxonomies of CSPs, creating new metrics for their evaluation, expanding a catalog of design patterns, among many other tasks that remain open for future work.

Acknowledgements

The authors would like to thank Jakob Rogstadius and Ji Kim Lucas for their contributions in the development of AIDR.

References

[1] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *Proc. of VLDB*, pages 81–92, 2003.

[2] O. Alonso, D. E. Rose, and B. Stewart. Crowdsourcing for relevance evaluation. In *ACM SigIR Forum*, volume 42, pages 9–15. ACM, 2008.

[3] B. H. Banathy. A taste of systemics. *The Primer Project*, 1997.

[4] 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 *Proc. of UIST*, pages 313–322. ACM, 2010.

[5] M. S. Bernstein, J. Brandt, R. C. Miller, and D. R. Karger. Crowds in two seconds: enabling realtime crowd-powered interfaces. In *Proc. of UIST*, pages 33–42. ACM, 2011.

[6] L. V. Bertalanffy. *General System Theory: Foundations, Development, Applications.* Geoohursdaorge Braziller, Inc, revised edition, 1969.

[7] J. P. Bigham, C. Jayant, H. Ji, G. Little, A. Miller, R. C. Miller, R. Miller, A. Tatarowicz, B. White, S. White, et al. Vizwiz: nearly real-time answers to visual questions. In *Proc. of UIST*, pages 333–342. ACM, 2010.

[8] C. Bockermann and H. Blom. Processing data streams with the rapidminer streams-plugin. In *Proc. of the RapidMiner Community Meeting and Conference*, 2012.

[9] I. Boutsis and V. Kalogeraki. Crowdsourcing under Real-Time constraints. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 753–764. IEEE, May 2013.

[10] A. Bozzon, M. Brambilla, S. Ceri, and A. Mauri. Reactive crowdsourcing. In *Proc. of WWW*, pages 153–164, 2013.

[11] K. Broda, K. Clark, R. Miller, and A. Russo. Sage: a logical agent-based environment monitoring and control system. In *Ambient Intelligence*, pages 112–117. Springer, 2009.

[12] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: a new class of data management applications. In *Proc. of VLDB*, pages 215–226, 2002.

[13] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *Proc. of SIGMOD*, pages 668–668. ACM, 2003.

[14] P. Dai, C. H. Lin, Mausam, and D. S. Weld. POMDP-based control of workflows for crowdsourcing. *Artificial Intelligence*, 202:52–85, Sept. 2013.

[15] H. Debar and A. Wespi. Aggregation and correlation of intrusion-detection alerts. In *Recent Advances in Intrusion Detection*, pages 85–103. Springer, 2001.

[16] G. Demartini, D. E. Difallah, and P. C. Mauroux. ZenCrowd: leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *Proc. of WWW*, pages 469–478. ACM, 2012.

[17] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Advances in Database Technology-EDBT 2006*, pages 627–644. Springer, 2006.

[18] J. L. G. Dietz and J. A. P. Hoogervorst. Enterprise ontology in enterprise engineering. In *Proc. of SAC*, pages 572–579. ACM, 2008.

[19] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *Communications of the ACM*, 54(4):86–96, 2011.

[20] J. Fan, M. Lu, B. C. Ooi, W.-C. Tan, and M. Zhang. A hybrid machine-crowdsourcing system for matching web tables. Technical report, National University of Singapore, 2013.

[21] T. Finin, W. Murnane, A. Karandikar, N. Keller, J. Martineau, and M. Dredze. Annotating named entities in twitter data with crowdsourcing. In *Proc. of the NAACL HLT 2010 Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk*, pages 80–88. ACL, 2010.

[22] A. Finnerty, P. Kucherbaev, S. Tranquillini, and G. Convertino. Keep it simple: Reward and task design in crowdsourcing. In *Proc. of Italian CHI*, 2013.

[23] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. In *Proc. of SIGMOD*, pages 61–72. ACM, 2011.

[24] J. D. Fraustino, B. Liu, and Y. Jin. Social media use during disasters: A review of the knowledge base and gaps (final report, start). Technical report, Human Factors/Behavioral Sciences Division, Science and Technology Directorate, U.S. Department of Homeland Security, College Park, MD, USA, 2012.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Nov. 1994.

[26] J. Gao, X. Liu, B. C. Ooi, H. Wang, and G. Chen. An online cost sensitive decision-making method in crowdsourcing systems. In *Proc. of SIGMOD*, pages 217–228. ACM, 2013.

[27] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the system S declarative stream processing engine. In *Proc. of SIGMOD*, pages 1123–1134. ACM, 2008.

[28] S. A. M. Geerts. Discovering crowdsourcing theory, classification and directions for use. Master's thesis, Eindhoven University of Technology, 2009.

[29] D. Geiger, M. Rosemann, and E. Fielt. Crowdsourcing information systems - a systems theory perspective. In *22nd Australasian Conf. on Information Systems*, Nov. 2011.

[30] G. Goel, A. Nikzad, and A. Singla. Allocating tasks to workers with matching constraints: Truthful mechanisms for crowdsourcing markets. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion*, WWW Companion '14, pages 279–280, Republic and Canton of Geneva, Switzerland, 2014. International World Wide Web Conferences Steering Committee. ISBN 978-1-4503-2745-9.

[31] J. Heer and M. Bostock. Crowdsourcing graphical perception: using mechanical turk to assess visualization design. In *Proc. of SIGCHI*, pages 203–212. ACM, 2010.

[32] C.-J. Ho and J. W. Vaughan. Online task assignment in crowdsourcing markets. In *AAAI*, 2012.

[33] H. Hodson. Online juries help curb bad gamer behaviour. *New Scientist*, (2912), Apr. 2013.

[34] J. J. Horton and L. B. Chilton. The labor economics of paid crowdsourcing. In *Proc. of ACM Electronic Commerce*, pages 209–218. ACM, 2010.

[35] A. L. Hughes and L. Palen. Twitter adoption and use in mass convergence and emergency events. *Int. Journal of Emergency Management*, 6(3):248–260, 2009.

[36] A. L. Hughes, S. Peterson, and L. Palen. Social media in emergency management. In J. E. Trainor and T. Subbio, editors, *Issues in Disaster Science and Management: A Critical Dialogue Between Scientists and Emergency Managers*. FEMA in Higher Education Program, 2014.

[37] M. Imran, S. Elbassuoni, C. Castillo, F. Diaz, and P. Meier. Practical extraction of disaster-relevant information from social media. In *Proc. of Workshop on Social Media Data for Disaster Management*, WWW '13 Companion, pages 1021–1024. ACM/IW3C2, 2013.

[38] M. Imran, S. M. Elbassuoni, C. Castillo, F. Diaz, and P. Meier. Extracting information nuggets from disaster-related messages in social media. In *Proc. of ISCRAM*, Baden-Baden, Germany, 2013.

[39] M. Imran, C. Castillo, J. Lucas, P. Meier, and S. Vieweg. Aidr: Artificial intelligence for disaster response. In *Proceedings of the companion publication of the 23rd international conference on World wide web companion*, pages 159–162. International World Wide Web Conferences Steering Committee, 2014.

[40] M. Imran, C. Castillo, J. Lucas, M. Patrick, and J. Rogstadius. Coordinating human and machine intelligence to classify microblog communications in crises. *Proc. of ISCRAM*, 2014.

[41] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. In *Proc. of Workshop on Human Computation*, pages 64–67. ACM, 2010.

[42] E. Kamar, S. Hacker, and E. Horvitz. Combining human and machine intelligence in large-scale crowdsourcing. In *Proc. of AAMAS*, pages 467–474. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

[43] G. Kazai. An exploration of the influence that task parameters have on the performance of the crowds. In *CrowdConf*, Oct. 2010.

[44] G. Kazai, J. Kamps, and N. Milic-Frayling. An analysis of human factors and label accuracy in crowdsourcing relevance judgments. *Information Retrieval*, pages 138–178, 2013.

[45] A. Kittur, J. V. Nickerson, M. Bernstein, E. Gerber, A. Shaw, J. Zimmerman, M. Lease, and J. Horton. The future of crowd work. In *Proc. of CSCW*, pages 1301–1318. ACM, 2013.

[46] A. Kulkarni, M. Can, and B. Hartmann. Collaboratively crowdsourcing workflows with turkomatic. In *Proc. of CSCW*, pages 1003–1012. ACM, 2012.

[47] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proc. of SIGMOD*, pages 2–11. ACM, 2003.

[48] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. TurKit: tools for iterative tasks on mechanical turk. In *Proc. of the ACM SIGKDD Workshop on Human Computation*, HCOMP '09, pages 29–30. ACM, 2009.

[49] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. CDAS: a crowdsourcing data analytics system. *Proc. of VLDB*, 5(10):1040–1051, 2012.

[50] I. Lykourentzou, D. J. Vergados, and V. Loumos. Collective intelligence system engineering. In *Proc. of MEDES*. ACM, 2009.

[51] I. Lykourentzou, D. Vergados, K. Papadaki, and Y. Naudet. Guided crowdsourcing for collective work coordination in corporate environments. In *Proc. of Computational Collective Intelligence*, volume 8083, pages 90–99. Springer Berlin Heidelberg, 2013.

[52] N. Marz and contributors. Storm project. `http://storm-project.net/`, 2013.

[53] A. J. Mashhadi and L. Capra. Quality control for real-time ubiquitous crowdsourcing. In *Proc. of workshop on ubiquitous crowdsourcing*, 2011.

[54] W. Mason and D. J. Watts. Financial incentives and the "performance of crowds". *SIGKDD Explor. Newsl.*, 11:100–108, May 2010.

[55] M. Mcgowran. Facebook outsources dirty-content clean-up to offshore moderators, March 2012.

[56] M. Mendes, P. Bizarro, and P. Marques. A performance study of event processing systems. In R. Nambiar and M. Poess, editors, *Performance Evaluation and Benchmarking*, volume 5895, pages 221–236. Springer Berlin Heidelberg, 2009.

[57] S. Muthukrishnan. *Data streams: Algorithms and applications.* Now Publishers Inc, 2005.

[58] Y. Naudet and I. Lykourentzou. Personalisation in crowd systems. In *(submitted to) the 9th Int. Workshop on Semantic and Social Media Adaptation and Personalization (SMAP2014)*, 2014.

[59] Y. Naudet, T. Latour, W. Guedria, and D. Chen. Towards a systemic formalisation of interoperability. *Computers in Industry*, 61(2):176 – 185, 2010.

[60] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proc. of ICWM Workshops*, pages 170–177. IEEE, 2010.

[61] J. Noronha, E. Hysen, H. Zhang, and K. Z. Gajos. Platemate: crowdsourcing nutritional analysis from food photographs. In *Proc. of UIST*, pages 1–12. ACM, 2011.

[62] A. Olteanu, C. Castillo, F. Diaz, and S. Vieweg. CrisisLex: A lexicon for collecting and filtering microblogged communications in crises. In *Proc. of ICWSM*, 2014.

[63] H. Om and A. Kundu. A hybrid system for reducing the false alarm rate of anomaly intrusion detection system. In *Proc. of RAIT*, pages 131–136. IEEE, 2012.

[64] A. Parameswaran, A. D. Sarma, H. Garcia-Molina, N. Polyzotis, and J. Widom. Human-assisted graph search: it's okay to ask questions. *Proc. of VLDB*, 4(5):267–278, 2011.

[65] A. J. Quinn and B. B. Bederson. Human computation: a survey and taxonomy of a growing field. In *Proc. of SIGCHI*, pages 1403–1412. ACM, 2011.

[66] J. Rogstadius, V. Kostakos, J. Laredo, and M. Vukovic. Towards real-time emergency response using CrowdSupported analysis of social media. In *CHI Workshop on Crowdsourcing and Human Computation: Systems, Studies and Platforms*, 2011.

[67] S. B. Roy, I. Lykourentzou, S. Thirumuruganathan, S. Amer-Yahia, and G. Das. Crowds, not drones: Modeling human factors in interactive crowdsourcing. *DBCrowd 2013*, page 39, 2013.

[68] S. Roy Chowdhury, M. Imran, M. R. Asghar, S. Amer-Yahia, and C. Castillo. Tweet4act: Using incident-specific profiles for classifying crisis-related messages. In *10th International ISCRAM Conference*, 2013.

[69] G. D. Saxton, O. Oh, and R. Kishore. Rules of crowdsourcing: Models, issues, and systems of control. *Information Systems Management*, 30(1):2–20, Jan. 2013.

[70] E. Schenk and C. Guittard. Towards a characterization of crowdsourcing practices. *Journal of Innovation Economics*, 2011.

[71] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *Proc. of the Third ACM International Conf. on Distributed Event-Based Systems*, page 4. ACM, 2009.

[72] A. Sorokin and D. Forsyth. Utility data annotation with amazon mechanical turk. In *Proc. of CVPR Workshops*, pages 1–8. IEEE, June 2008.

[73] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[74] P. sudhakar Lahane. Data stream mining. *International Journal of Advances in Computing and Information Researches*, 1(1):6–10, 2012.

[75] A. Sun, D. D. Zeng, and H. Chen. Burst detection from multiple data streams: a network-based approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 40(3):258–267, 2010.

[76] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of VLDB*, pages 309–320, 2003.

[77] D. Turaga, H. Andrade, B. Gedik, C. Venkatramani, O. Verscheure, J. D. Harris, J. Cox, W. Szewczyk, and P. Jones. Design principles for developing stream processing applications. *Software: Practice and Experience*, 40(12):1073–1104, 2010.

[78] L. Von Ahn and L. Dabbish. Labeling images with a computer game. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 319–326. ACM, 2004.

[79] M. Vukovic. Crowdsourcing for enterprises. In *Services - I, 2009 World Conf. on*, volume 0, pages 686–692. IEEE, July 2009.

[80] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *Proc. of SIGMOD*, pages 229–240. ACM, 2013.

[81] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. In *Proc. of VLDB*, 2013.

[82] X. S. Yang, D. W. Cheung, L. Mo, R. Cheng, and B. Kao. On incentive-based tagging. In *Proc. of ICDE*, pages 685–696. IEEE Computer Society, 2013.

[83] M.-C. Yuen, I. King, and K.-S. Leung. A survey of crowdsourcing systems. In *Proc. of the IEEE Third International Conf. on Social Computing (SocialCom)*, pages 766–773. IEEE, Oct. 2011.

[84] P. Zhang, B. J. Gao, P. Liu, Y. Shi, and L. Guo. A framework for application-driven classification of data streams. *Neurocomputing*, 92:170–182, 2012.

[85] A. Zhou, F. Cao, W. Qian, and C. Jin. Tracking clusters in evolving data streams over sliding windows. *Knowledge and Information Systems*, 15(2):181–214, 2008.

[86] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proc. of VLDB*, pages 358–369, 2002.