2011 15th IEEE International Enterprise Distributed Object Computing Conference

# Resource and Agreement Management in Dynamic Crowdcomputing Environments

Harald Psaier, Florian Skopik, Daniel Schall, Schahram Dustdar
*Distributed Systems Group*
*Vienna University of Technology*
*Argentinierstraße 8/184-1, A-1040 Vienna, Austria*
{lastname}@infosys.tuwien.ac.at

***Abstract*—Open Web-based and social platforms dramatically influence models of work. Today, there is an increasing interest in outsourcing tasks to crowdsourcing environments that guarantee professional processing. The challenge is to gain the customer's confidence by organizing the crowd's mixture of capabilities and structure to become reliable. This work outlines the requirements for a reliable management in crowdcomputing environments. For that purpose, distinguished crowd members act as responsible points of reference. These members mediate the crowd's workforce, settle agreements, organize activities, schedule tasks, and monitor behavior. At the center of this work we provide a hard/soft constraints scheduling algorithm that integrates existing agreement models for service-oriented systems with crowdcomputing environments. We outline an architecture that monitors the capabilities of crowd members, triggers agreement violations, and deploys counteractions to compensate service quality degradation.**

***Keywords*-crowdsourcing, service agreements, scheduling, behavior-based adaptation**

## I. Introduction

Crowdsourcing applications [1] are typically open Internet based platforms where problem-solving tasks are distributed among a group of humans. Crowdsourcing follows the 'open world' assumption, allowing humans to provide their capabilities through the platform by registering themselves as members. A popular crowdsourcing platform is for example Amazon Mechanical Turk [2]. Currently, two types of crowdsourcing modes have been identified [3]. In marketplace oriented crowdsourcing crowds are organized by providers or brokers that bid for and distribute requests. In competition based crowdsourcing the request is an open call to the crowd and the winning submission is picked.

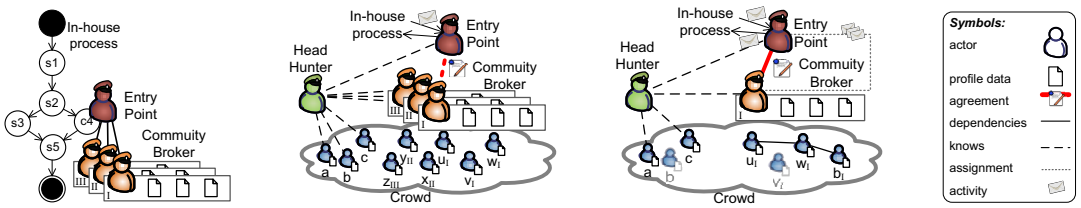
While conventional enterprise systems rely on well established policies, crowdsourcing has a more loosely-coupled, dynamic, and flexible structure and depends especially on the preferences and behavior of the individual crowd members. Even if an advantage of a crowd platform is the possibility to choose from a larger number of skilled members, the selection must consider, e.g., the distinguished working hours of the members possibly contradicting with the current requirements. The members availability will mostly depend on their context. Their working hours, for example, also depend on their location and the related timezone. Context does not only influence task assignment strategies but certain changes can also cause unpredictable interruptions of services. This leads to an incomplete and unsatisfactory task state at deadline. As a consequence, meeting promised service contracts is challenging and demands for sophisticated management techniques for a crowd platform. One of the key issues of the management investigated in this work is to find an appropriate scheduling for task assignments that matches tasks to skills and to the availability of the members, and above all, also meets the agreed contract. This adaptive scheduling strategy must constantly update on changes, and ensure, that shifts in interest, skills, and behavior of members including task-related misbehavior, such as degrading worker performance, refusal of tasks, or missing feedback that affects successful task completion is detected, avoided, and balanced with alternative workforce.

In this paper we describe a framework which integrates agreements into service-oriented crowdsourcing platforms. The prerequisite is a monitoring infrastructure that updates a crowd-based resource model. In our previous work [4], [5], we studied a monitoring and behavior adaptation architecture serving as the basis for the presented agreement management. Here, we focus on an agreement model combined with an adaptation approach for reliable task execution. An accurate resource model supports a sophisticated scheduling of crowd activities. A self-adaptive mechanism must anticipate misbehavior and update the crowd's task scheduling to enforce behavior rules also dictated by the agreement.

Here we address the following challenges:

- *Human Behavior Characteristics.* The crowd is a transient network where humans can join and leave the platform at any time. Furthermore, in contrast to software services, human behavior is subject to numerous contextual constraints.
- *Feedback Loop.* Since the crowd's resources, i.e., available members and their capabilities, are in a constant flux and change, models for monitoring the environment and accounting for given constraints need to be applied.
- *Quality Guarantees.* It is challenging to provide any guarantees regarding execution time and reliability of actors in highly dynamic environments. Our flexible agreement management models tackle that problem by allowing management of negotiated agreements.

1541-7719/11 $26.00 © 2011 IEEE
DOI 10.1109/EDOC.2011.9




(a) Test cycle. (b) Agreement setup. (c) Assignment. (d) Symbols.

Figure 1. Crowdsourcing software tests.

This paper is structured as follows: in Section II related work is discussed followed by a section describing a crowdcomputing scenario. Section IV details a framework that addresses identified challenges regarding agreement management. Section V discusses the layout of agreements and use of quality metrics for adapting crowd scenarios in order to meet agreements. One promising adaptation approach is dynamic re-scheduling of tasks in crowds which is highlighted in Section VI. Section VII shows evaluation results and Section VIII concludes the paper.

## II. Related Work

**Crowdsourcing** applications [1] are online, distributed problem-solving and production models that have emerged in recent years ([2], [6], [7]). A vast number of registered individuals offer solutions to the problem. Apart from the benefit of multiple, redundant workforce and collective intelligence, crowdsourcing poses some difficult challenges related to its distributed and open nature. The main challenge remains how to organize and manage the crowd and identify potential leaks of resources. Two operating modes, marketplace and competition based crowd platforms, have been identified so far [3]. Our assumptions base on a marketplace oriented crowdsourcing environment where mediators manage the crowd. Moreover, we base our ideas around existing mediator services that post tasks on behalf of their clients [8].

There has been substantial research on translations of **service level agreements** to a Web-service applicable standard. Two of the main contributions are the specification from the Grid Resource Allocation Agreement Protocol (GRAAP) Working Group [9] and from IBM [10]. Both present similar XML-based model for an SLA, however, differ in the details. When creating their specification the GRAAP Working Group in particular focuses on the setup, negotiation, and renegotiation phases of the agreement, thus, presents a rather flexible structure [11]. IBM's WSLA focus was on defining agreement objectives, their constraints and combination. For this purpose parameters can be linked to SLOs together with thresholds. In our work we reuse the parameter scheme to define our quality attributes.

**Scheduling** is a well-known subject in computer science. The novel contribution in this work is to consider multidimensional assignment and allocation of tasks. A thorough analysis and investigation in the area of multidimensional optimal auctions and the design of optimal scoring rules has been done by [12]. In [13] iterative multi-attribute procurement auctions are introduced while focusing on mechanism design issues and on solving the multi-attribute allocation problem. Focusing on task-based adaptation, [14] near-optimal resource allocations and reallocations of human tasks were presented. Workforce scheduling research investigates the impact of weights on the multiple, at times contradicting, objectives in real work workforce scheduling [15]. Staff scheduling related to closed systems was discussed in [16], [17]. However, unlike in closed enterprise systems, crucial scheduling information, i.e., the current user load or precise working hours are usually not directly provided by the crowd. Instead, the scheduling relevant information must be gathered by monitoring. The work in [18] details the challenges for collaborative workforce in crowdsourcing where activities are coordinated, workforce contributions are not wasted and final results are guaranteed.

## III. Crowdcomputing Environment

This section motivates our work and outlines a crowdcomputing environment. With reference to existing testing marketplaces, c.f. [19], we present a software testing scenario and discuss the different roles of crowd members.

### A. Roles in Crowdcomputing

The crowd *Entry Point(EP)* is a mediator that connects customers from outside the crowd with the required crowd members (see Figure 1). Generally, crowd customers look for a certain knowledge or capability which their company environments lack. Thus, the EP maintains regular contacts to required crowd members. It acts as representative of a company's outsourced assignments to the crowd and must assure all implications to the dependencies with the company's internals. The *Community Broker (CB)* is a proxy for a certain crowd segment or platform. It maintains and represents a group of registered members with similar capabilities and offers the joint knowledge to interested parties, e.g., EPs. In the example of Figure 1, the representative of platform $I$ is the CB of the crowd member group $u$, $v$ and $w$ with a $I$ subscript each. A CB is in charge of a fair distribution of the incoming assignments and settles agreements. The *Head Hunter (HH)* acts as a kind of registry

for a crowd environment. It monitors the tendencies of the required capabilities in the crowd environment, and discovers new knowledge sources (single members or groups). Additionally, it offers an interface that allows new members to register, and further, to be discovered. EPs and CBs can find required partners and members using the HH's lookup service. In the example in Figure 1, the HH provides, e.g., for the new crowd members $a$, $b$, and $c$ a first point of reference to enter the crowd business. We argue that, between entities of these roles, relations based on agreements also need to be established. Agreements state rules that organize and regulate the assignments of tasks in the crowd. They help to assure dependencies and regulate the rather unpredictable behavior of an unorganized crowd.

### B. Use Case Scenario

In Figure 1 we outline the various phases of a typical crowdsourcing software test scenario. Beginning with Figure 1(a) the in-house quality assurance (QA) process is split into five repetitive and automated steps (*s1*, *s2*, *s3*, the crowdsourcing step *c4*, and *s5*). In *s1* all modules that need to be tested are collected for the next upcoming QA cycle. In *s2* a sorting process differs between automated tests that run in the company's own test environment and those that need to be crowdsourced by, e.g., monitoring a flag on the test units provided by the QA team. Step three (*s3*) represents the test period run in-house. In parallel, in step *c4* a company's representative, EP, is in charge of a smooth flow of the crowdsourced test activities. In order to get into this position, s/he needs to have some previously established relations to available crowd platforms, e.g., *I*, *II*, and *III*, and their representative CBs. Having numerous alternatives for outsourcing guarantees a reliable management of this test period. The final step *s5* collects the testing results for an evaluation and merge.

Next, in Figure 1(b), the EP has to decide which crowd community can handle the currently pending test activities. The assignment depends on the requirements of the tasks and resources of current platforms, in particular, the members' capacities and capabilities. Next, the EP must balance the effort, expected quality, and costs of the available CBs' offers. If none of the known CBs fits the requirements, the EP can invoke the HH lookup service to find a new CB. Thus a HH mediates CBs to an EP on request. In the example, however, this is not the case and EP contacts the chosen platform $I$'s representative. An agreement for the following assignment is negotiated.

Figure 1(c) illustrates the scenario at runtime. Testing activity segments, i.e. tasks, are delegated to the appropriate crowd members. As mentioned in the introduction, the crowd's structure is transient and its members' processing attitude is context dependent and individual, thus, at times unpredictable. In the given example, nodes $u$, $v$, and $w$ process dependent tasks. However, $v$ is not able to process the tasks as scheduled. It is now the challenge of the crowd management to find a solution. A first solution would be to reschedule the task. Unfortunately, in the presented scenario no member can replace $v$. The CB must call the assistance of the HH and request a fitting, though previously unknown, crowd member. As depicted, member $b$ is mediated to CB of platform *I* and takes over the duties of node $v$. Finally, once all tasks have been completed, the results are collected and merged by the EP. Thereafter, the final result is provided to the evaluation step (*s5*).

### C. Agreement Management

A CB is usually either a business person who established a dedicated platform and invited crowd members to join, or, has emerged from a formation of members with the same interests to represent them. Because crowd environments are open systems with no guarantees, the main role of the CB is to fill this gap. S/He provides the otherwise missing, however, necessary guarantees to the EP. In particular, guarantees in this scenario include a satisfying, proper conduct of the tests. Just like in a cooperation between two companies, EP and CB, set-up an agreement for the outsourced test activity. The agreement identifies the activity, settles the test scheduling, and states metrics to measure the demanded quality. The quality preferences include attributes, such as, maximum tolerated running time for the assigned tasks, fees, demands on the result, etc..

Our proposed agreement management approach employs the following fundamental concepts when distributing tasks in the crowd:

*Hard- and Soft-Constraints.* We distinguish between criteria that must be met, e.g., expertise area of crowd members and their principal participation interest and so-called soft constraints that are used for ranking potential crowd members, including their capacity, reputation, and costs.

*Environment Observation.* Periodic run-time monitoring and evaluation of the crowd members' behavior in terms of reliability and task execution progress enables timely detection of misbehavior and quality degradations.

*Adaptation and Optimization.* Using feedback data obtained from behavior monitoring enables numerous adaptation mechanisms to optimize the assignment and execution of tasks in the crowdcomputing environment; for instance the reassignment and/or rescheduling of tasks in case of deadline misses.

## IV. Architecture

The new extensions to our previous work on the $VieCure\ Framework$ [5] are detailed in the following section. The additions include monitoring of agreements and scheduling of assignments based on policies.

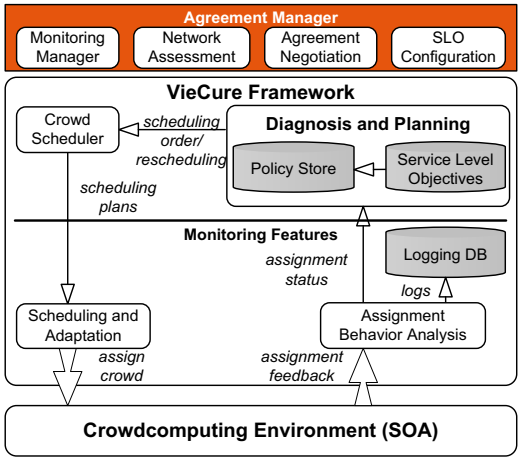



Figure 2. Agreement management framework.

### A. Architectural Overview

Figure 2 outlines the three-layer infrastructure of the framework extension. The top layer comprises the *Agreement Manager*. This is a tool-set to monitor, track, and analyze the crowd structure. Additionally, by hiding the particularities of the scheduling technique, it allows to extend and change the framework's *SLOs* and *Policy Store* entries, thus, adapt the environment to new agreements. The layer in the middle represents the framework itself. It is organized according to an **automated adaptation loop** and its main purpose is to adapt the *Crowd Scheduler*'s assignment strategy. The strategy depends on the current crowd's acceptance behavior and capacities, as well as, on policies representing system and agreement constraints. Therefore, interfacing with the environment, the *Assignment Behavior Analysis* collects feedback to a log database and forwards the current status to the *Diagnosis and Planning* module. Considering the valid policies and the fresh assignment status from analysis this module adapts the scheduling order, and/or, on an assignment reject, issues a rescheduling directive with new ordering rules to the *Crowd Scheduler*. Depending on the current situation the *Crowd Scheduler* uses its algorithm to assign a batch of tasks, or on request of *Diagnosis and Planning* module, reschedules a unsuccessful assignment. Finally, the scheduling result is transmitted to the *Scheduling and Adaptation* module. This deploys the assignments and scheduling changes to the crowd.

### B. Further Building Blocks

The bottom layer of the architecture in Figure 2 bases on a *service-oriented architecture (SOA)*. In our previous work, we studied flexible interactions [20], metrics in crowds [21], and monitoring of service-oriented collaboration environments [5]. Here we give a quick overview of the main principles and our findings.

**SOA-based Interactions in Crowds.** Dynamic discovery of services, flexible interactions and compositions at runtime are only some properties that make SOAs an intuitive and convenient technical grounding for large-scale crowdsourcing environments. However, not only service interactions, but also human interactions may be performed using SOAP (see Human-Provided Services [20] for collaborative environments and BPEL4People [22] for human interactions in business processes), which is the state-of-the-art technology to exchange XML-based messages in service-oriented environments, and well supported by a wide variety of software frameworks. There exist well-known and accepted standards for modeling and monitoring service level agreements that act as an integral part of our approach (c.f., Section II).

**Failure Compensation through Dynamic Adaptation.** Performance degradation and failures may arise due to various reasons. Especially in crowdsourcing environments, human (mis-)behavior has a fundamental influence on the overall success rate regarding task execution and throughput. We studied an approach to rate and categorize human behavior earlier [5] to be able to compensate malicious behavior in collaborative networks. For that purpose, typically adaptation rules are pre-defined (e.g., seize tasks from unreliable workers) and applied according to adaptation policies. These mechanisms rely on monitoring data that is captured from the service-oriented infrastructure.

The next section gives a detailed description of the sequence of operations between the framework's modules and outlines their interaction with the system in the various phases of an agreement's life-cycle.

### C. Agreement Life-Cycle

The life-cycle of the agreement includes three distinct phases. In the first phase, offers are invited and an agreement for the assignment is negotiated. With the agreement as a base for the business relation, the additional environment policies need to be applied in the line with the agreement. Next, tasks are scheduled and assigned to the crowd members according to the policies order. In parallel, the assignments status is diagnosed. A rejected assignment must be rescheduled.

The sequence in Figure 3 presents an agreement's life-cycle. It details the interactions between the involved parties, EP and CB presented in Section III, and the automated $VieCure$ crowd assignment management.

**Negotiation.** While an EP browses for interesting bids, the CB uses the $VieCure\ Framework$ to create offers within the limits of the current crowd's capacities. On a request an individual offer can be created and provided. The negotiable items of a later assignment and their boundaries depend on the available resources and their current scheduling. Both can be gathered by checking availability at the *Crowd Scheduler* and the current crowd member status at the *Diagnosis and Planning* module. The provided offer is revised by the

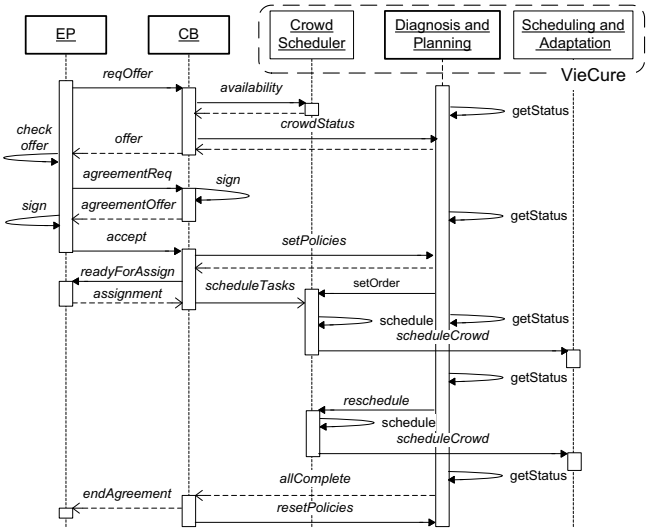



Figure 3. Agreement management life-cycle.

EP comparing it against the in-house requirements. If not pleased, negotiation starts over again or a different CB is considered as service provider. Finally, a satisfying offer including the agreement details in the objectives is signed by both parties and enacted with the according policies for assignment management.

**Scheduling.** Once the agreement's objectives are translated into new scheduling policy rules, the CB is ready to take over the assignment and schedule the tasks in sequence. Meanwhile, tasks are arranged by the *Crowd Scheduler*'s strategy according to a valid order and their priorities. The scheduling plan is propagated to the *Scheduling and Adaptation* module. Then, the in sequence distribution of the tasks to the members concludes the scheduling phase.

**Rescheduling.** Situation and behaviors change. Some of the members will reject their scheduling plans. The *Diagnosis and Planning* module receives the current assignment status of all involved members and reacts to rejects with rescheduling orders for the *Crowd Scheduler*. As the members' status has changed with previous assignments, the *Diagnosis and Planning* module must also adapt the scheduling strategy for any reassignment. Generally, it keeps a record of the rejects and accepts and adapts the present scheduling strategy to the current crowd's acceptance behavior.

In the line with the requirements emerging from the agreement's life-cycle in the next section we detail a structure for the agreements and discuss examples of fundamental quality attributes applicable for crowdcomputing.

## V. Agreements and Quality

The growing interest in outsourcing tasks to platforms hosting crowds entails the demand for reliable business contacts, clear rules, and applicable agreements. A prerequisite of our approach is a reliable behavior monitoring. This ensures up-to-date data for metrics and quality attributes.

### A. Agreement Structure

The used agreement model is inspired by the work of IBM and the GRAAP Working Group presented in Section II. The overall structure includes header, agreement items, and terms. The header comprises the agreement's parties details and contact information. In the contractual items the agreement's subjects are listed. These include the service content (i.e., in Web-service environments WSDL location, endpoint, and operation) along with scheduling information, metrics and their measuring method. Finally, the terms provide the objectives, SLOs respectively, and their validity period. Threshold values expresses the desired relation between objectives and metrics defined in the items. In the next section, we provide a number of quality metrics that can be applied and measured in SOA based crowdcomputing environments, and thus, aligned to the described structure.

### B. Quality Parameters

Analyzing the requirements of the scenario in Section III, Table I represents a plausible list of quality attributes for negotiation in crowdcomputing. The crowd broker manages the attributes for registered crowd members and constantly updates their value.

Table I
Negotiated quality attributes.

| Quality Attributes | Description |
|---|---|
| reliability | predicted confidence in the assignment acceptance of a member. |
| load | estimated task queue size of a member. |
| overlap | match between a member's capabilities and the task's requirements. |
| cost | fee demanded by a member for processing a task. |

The first listed quality attribute, *reliability*, is related to the assignment acceptance behavior of a particular member. It reflects the difference between total assigned and rejected tasks. The monitored *load* represents the current task load at a member. The *overlap* factor indicates how suitable a member is for a certain task assignment by calculating the overlap of its capabilities and the task's requirements. Lastly, the *cost* attribute states the maximum fee that can be charged by a member for a processed task.

## VI. Task Scheduling in the Crowd

Next, we provide an example of an agreement in extended WSLA (Web Service Level Agreements)[1] notation. The format is XML-based, thus, processable for the phases of negotiation, and extraction of agreement items and objectives. Further, this helps to fit the extracted hard- and soft-constraints into a self-adaptive scheduling algorithm that is formalized in Algorithm 1.

[1] http://www.research.ibm.com/wsla/WSLA093.xsd

### A. Agreement Setup

The XML examples in Listing 1 and Listing 2 detail a sample WSLA agreement applicable to crowdcomputing environments. Only the important parts are fully listed. Additionally, the structure has been extended to fit the crowdcomputing particulars.

```
1  <wsla:SLA
2  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3  xmlns:wsla="http://www.ibm.com/wsla"
4  xmlns:hps="http://www.infosys.tuwien.ac.at/hps/"
5  name="SwTestCrowdSLA5312">
6   <wsla:Parties>
7   <wsla:ServiceProvider name="CommunityBroker">
8    <!-- ... -->
9   </wsla:ServiceProvider>
10  <wsla:ServiceConsumer name="EntryPoint">
11   <!-- ... -->
12  </wsla:ServiceConsumer>
13 </wsla:Parties>
14 <wsla:ServiceDefinition name="TestService">
15  <wsla:Operation
16   xsi:type="wsla:WSDLSOAPOperationDescriptionType"
17   name="AddActivity">
18   <!-- schedule period -->
19   <wsla:SLAParameter name="FeedbackExpected"
20    type="int" unit="Days">
21    <wsla:Metric>CountDays</wsla:Metric>
22   </wsla:SLAParameter>
23   <!-- config details: name, wsdl-location, binding -->
24  </wsla:Operation>
25  <hps:SLAActivity name="Testing">
26   <hps:SLAInvolvedProfiles>
27    <hps:SLAProfileTyp>UI Test</hps:SLAProfileTyp>
28    <!-- ... functional, performance, security -->
29   </hps:SLAInvolvedProfiles>
30   <hps:SLAReportURI>http://.../reports</hps:SLAReportURI>
31   <wsla:SLAParameter name="TasksCost"
32    type="float" unit="Euro">
33    <wsla:Metric>MaxCost</wsla:Metric>
34   </wsla:SLAParameter>
35     <!-- reliability, load, overlap -->
36  </hps:SLAActivity>
37 </wsla:ServiceDefinition>
38 <!-- wsla Obligations -->
39 </wsla:SLA>
```

Listing 1. Involved parties and body.

After the contract parties' details, *ServiceProvider* and *ServiceConsumer* listed in lines 6 to 13, Listing 1 states the items from line 14 to 37. These are a collection of *ServiceObjectType* items including scheduling, operation description, and configuration, and also, an *SLAParameter* (`FeedbackExpected`) arranging periodic feedback. Important and a new contribution to this part is the *SLAActivity* (lines 25 to 36). It extends WSLA's *ServiceObjectType* and states what kind of member capabilities must be involved in the testing activity (line 27), at which URI the final testing reports are expected (line 30), and at the end, the *SLAParameter*s for the assigned activity.

These include for example, all the quality attributes as presented previously in Table I. The parameter is identified by a name, a type, a unit, and related to a metric for estimation.

Listing 2 shows the terms as *Obligations* of the contract including all SLOs. An SLO consists of an obliged party, a validity period, and *Expression*s that can be combined

```
1  <wsla:Obligations>
2  <wsla:ServiceLevelObjective name="sloSrv"
3      serviceObject="AddActivity">
4   <wsla:Obliged>CommunityBroker</wsla:Obliged>
5   <!-- Validity -->
6   <wsla:Expression>
7    <wsla:Predicate xsi:type="wsla:Equal">
8     <wsla:SLAParameter>FeedbackExpected</wsla:SLAParameter>
9     <wsla:Value>7</wsla:Value>
10   </wsla:Predicate>
11  </wsla:Expression>
12 </wsla:ServiceLevelObjective>
13 <wsla:ServiceLevelObjective name="sloAct"
14     serviceObject="Testing">
15  <wsla:Obliged>CommunityBroker</wsla:Obliged>
16  <!-- Validity -->
17  <wsla:And>
18  <wsla:Expression>
19   <wsla:Predicate xsi:type="wsla:LessEqual">
20    <wsla:SLAParameter>TasksCost</wsla:SLAParameter>
21    <wsla:Value>50.0</wsla:Value>
22   </wsla:Predicate>
23  </wsla:Expression>
24  <!-- expressions for reliability, load, overlap -->
25  </wsla:And>
26  <wsla:EvaluationEvent>TaskAssignment</wsla:EvaluationEvent>
27 </wsla:ServiceLevelObjective>
28 <wsla:QualifiedAction>
29  <wsla:Party>CommunityBroker</wsla:Party>
30  <wsla:Action actionName="violation" xsi:type="Notification">
31   <wsla:NotificationType>Violation</wsla:NotificationType>
32   <wsla:CausingGuarantee>sloAct</wsla:CausingGuarantee>
33   <wsla:SLAParameter>MaxCost</wsla:SLAParameter>
34   <!-- expressions for reliability, load, overlap -->
35  </wsla:Action>
36 </wsla:QualifiedAction>
37 </wsla:Obligations>
```

Listing 2. Obligations and SLOs.

with a logic expression (e.g., *And*). The content of the expressions connects the pool of *SLAParameters* of the items to a predicate (e.g, `Equal`) and threshold value (*Value*). The final tag *QualifiedAction* defines the consequence of an SLO violation. In the example case, if a threshold of SLO `solAct` is violated an action *Notification* is called.

### B. Scheduling Algorithm

As mentioned in the introduction there is numerous factors that influence the human behavior. Thus, one crucial factor when scheduling tasks in crowdcomputing environments is that one cannot rely on the constant availability of resources (i.e., humans). These dynamics and the system size inhibit a fully automated scheduling approach. Instead in this work we base our assumptions on a semi-automated task assignment algorithm with a human, e.g. CB, in-the-loop. Such an approach remains highly adaptable and suitable for crowds.

**Tasks** that are outsourced to the crowd have three important categories of properties. First, keywords describe the task's type and required capabilities. These are matched to the members' profiles. Second, a task has temporal constraints for the scheduling process. A task has a *latest begin time*, and a *length* as the estimated time spent to complete the task. Combined, these properties define the *deadline*, and also, the latest possible *reassign time*. A task assignment fails if members do not acknowledge processing until the task's latest begin time. In many scenarios including soft-

ware testing tasks depend on one another. Thus, a property of a task can either represent a parent task with dependent subtasks, a subtask or an independent task. The impact of a failed processing of a parent task results also in a failure of the dependent subtasks. This needs to be considered when scheduling complex tasks in crowdcomputing environments.

In the following we detail the steps of the crowd scheduling algorithm. Let us define the set of crowd members $U = \{u_1, u_2, \ldots\}$ and the set of tasks $T = \{t_1, t_2, \ldots\}$ to be processed by the crowd. The goal of the algorithm is to assign each task to one individual crowd member. Notice, we do not make any assumption about the role of the broker. In fact, the broker may be implemented as a software service, thus the procedure is fully automated, or the procedure may be performed under human supervision.

**Algorithm 1** Task scheduling in the crowd.

```
Require: T ≠ ∅ ∧ U ≠ ∅
 1  AT ← ∅ /* Set of assigned tasks */
 2  FT ← ∅ /* Set of failed (to assign) tasks */
 3  foreach Task t ∈ T do
 4     /* Retrieve matching members with U' ⊆ U */
 5     U' ← getAllMembersMatchingTask(t)
 6     foreach Member u ∈ U' do
 7        /* Check additional constraints */
 8        if meetsDeadline(u, t) == false ∨
           meetsResponseTime(u, t) == false then
 9           continue /* Do not consider as candidate */
10        end if
11        score_(u,t) ← calculateScore(u,t)
12        U'' ← insertByScore(score_(u,t), U'')
13     end foreach
14     CM[t] ← U'' /* Save CM */
15     foreach Member u ∈ CM[t] do
16        if assignTask(u,t) == true then
17           AT ← AT ∪ t
18           break
19        else
20           CM[t] ← CM[t]\u /* Remove u from CM */
21        end if
22     end foreach
23     if t ∉ AT then
24        FT ← FT ∪ t /* Add t to FT */
25     end if
26  end foreach
```

Given the loop in Algorithm 1 (lines 3 to 26), three essential steps are performed:

**Matching.** A set of members whose profiles satisfy a task's required capabilities (see line 5) are selected. The detailed profile matching procedure is, however, not detailed in this work. Also, the demanded degree of match depends on the nature of a task (parent or subtask). In our system, parent tasks demand for a broader area of expertise thereby requiring a full match of a task's keywords and a members capabilities.

**Ranking.** Next (see lines 6 to 13) additional filtering and ranking is performed. The steps are a mixture of hard- and soft constraints. First, `meetsDeadline` is a filter to evaluate a task's deadline against the user $u$'s expertise profile and estimated load. An expert who has already proven experience collected by processing a given type of tasks may finish a task faster compared to a relatively unexperienced user. Also, the estimated current load of a user from a particular broker's point of view is taken into account; i.e., whether or not $u$ will be able start processing a task without violating time constraints such as deadline. Second, the filter `meetsResponseTime` is based on the user's context. Crowd members may be scattered around the globe. Therefore, different timezones as well as preferred working hours may prevent a member from processing a task in a given time frame. These two filters rely on hard constraints and cannot be influenced. The soft constraints are covered by a third type procedure. It performs a ranking based on a set of metrics that are specified in the context of an agreement. The details regarding this step (line 11) are given in the following.

**Assignment.** Finally, based on the ranked candidate list $CM[t]$ (see line 14), the broker attempts to assign the task $t$ (see line 16). Indeed, an assignment may fail due to the aforementioned challenges in crowdcomputing such as limited knowledge of the user's actual load. If the assignment succeeds, the task $t$ is added to the set $AT$ and our algorithm continues to process the next task. Otherwise, the member $u$ is removed from the list of candidate members $CM[t]$ (see line 20). Notice, the list $CM[t]$ is kept for reference in case a given task $t$ needs to be seized from the assigned member due to lack of processing progress. In this case, the next (top-ranked) member in $CM[t]$ is picked. The set of failed tasks $FT$ may require renegotiation of agreement metrics. Renegotiation procedures are currently not covered by our approach. The scoring function used in our algorithm (line 11) is defined as

$$score_{(u,t)} = \left[ \sum_{m \in M'} |w_m| \times score(u,m)^p \right]^{1/p} \quad (1)$$

The detailed parameter description can be found in Table II. This approach is based on a model for *simultaneity* and *replaceability* of preference parameters known as Logic-Scoring of Preferences (LSP), e.g. [23].

The parameter $p$ can be assigned manually based on the desired scoring behavior [23] or calculated automatically. Here we use a simple pattern to calculate $p$ based on the homogeneity (or diversity) of the preference weights $w_m \in W$ where $W$ is the set holding preference weights for each metric $m$: if $\texttt{max}(W) - \texttt{min}(W) > \texttt{avg}(W)$ use $p = 1.5$, if $\texttt{max}(W) - \texttt{min}(W) = \texttt{avg}(W)$ use $p = 1$, otherwise use $p = 0.5$. This means that replaceability should be preferred over simultaneity if the weight values (preferences) vary highly expressed by the relationship between `max` and `min` values compared to the average (`avg`) weight.

Table II
DESCRIPTION OF RANKING PROCEDURE.

| Symbol | Description |
|---|---|
| $m$ | Metric $m \in M'$ with $M' \subseteq M$ as defined in the SLA. Examples of a set of metrics and corresponding values that are obtained through monitoring and mining include *reliability* and *load*. |
| $w_m$ | The weight assigned to a given metric $m$ such that $\sum_{m \in M'} \lvert w_m \rvert = 1$. Weights are thereby not assigned independently or arbitrarily but rather with respect to preferences for individual metrics. |
| $score(u, m)$ | Scoring function for a given user $u$. The sign of a weight $w_m$ is used to determine whether higher or lower values denote better scores. For example, higher reliability results in higher scores (i.e., $+w_{rel}$) whereas lower values in costs are more desirable (i.e., $-w_{cost}$). |
| $p$ | Parameter to configure *simultaneity* or *replaceability* of a metric. Simultaneity is a desired property if *each* metric $m$ is important. As an example, a member should have good scores for *both* overlap and reliability as opposed to having only good overlap. Replaceability means that higher overlap may compensate for low reliability or vice versa. |

## VII. SIMULATION AND EVALUATION

The main idea of the evaluations is to identify the boundaries of the scheduling algorithm presented in Section VI in crowdcomputing environments to support reasonable negotiations of quality attributes in agreements. Monitored with the metrics defined by common crowdsourcing quality parameters presented in Section V we setup a simulated crowdcomputing environment with properties related to the environment outlined in the scenario. We consider only a subgroup of a larger and more complex crowdcomputing network because in contrast to existing environments with our broker approach resources can be organized for specific skills. In particular, we study the challenges and effort of scheduling, and also of rescheduling, tasks for differently behaving crowd members.

### A. Environment Setup

The **simulated crowd environment** comprises a framework implemented in Java language with a CB singleton instance, a crowd of 128 members, a task model, and various helper instances for the score calculation as detailed in the previous section. The single CB holds a reference to all crowd members in a registry. In a loop it tries to schedule batches of tasks for the members according to our scheduling algorithm and to reschedule tasks if rejected.

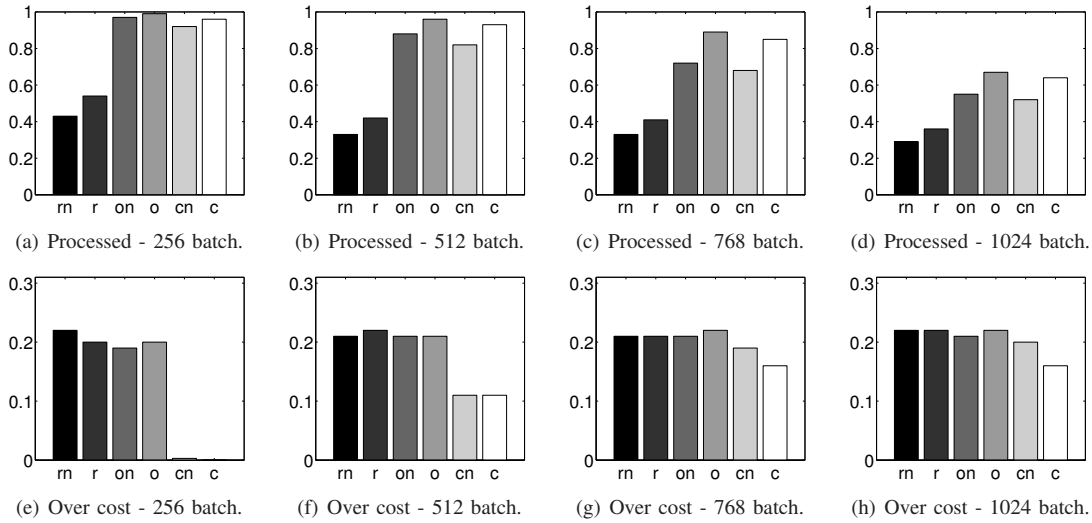
Each **crowd member** has its own capability profile. Additionally, the member exposes a predefined behavior in task assignment and task processing. The acceptance behavior in task assignment depends on the current task queue size. Whilst on an empty queue the member is eager to get task assignments, the enthusiasm decreases linearly to full reject at a number of 6 tasks in queue. The processing behavior is assigned at bootstrapping. Three different types of behavior patterns are known to the system. The first one processes tasks with a probability of 20%, the second one 50%, and the last one 80%. The behavior patterns are equally distributed among the members. Finally, members charge a fee for their service. In a further extension, a quarter of the members are randomly assigned with a fee rate which we consider to exceed the fixed rate negotiated in the agreement.

**Tasks** have temporal properties as discussed previously. Tasks have a latest begin time and length in time slices. In our experiments reassign time was set to 3 slices prior to latest begin. If an assigned member does not acknowledge processing until the task's latest begin time, the task fails. The impact of a failed processing of a complex (parent) task results also in a failure of the dependent subtasks.

At bootstrap the framework instantiates broker and members and fills the registry. The runtime is split in two phases including **scheduling and rescheduling** activities. Both phases apply Algorithm 1 to assign a batch of tasks. Members have different behavior in accepting and processing tasks. Thus, the goal of the two phases is to minimize the task failure rate, and in the later experiments, to minimize the costs of processing by choosing the currently less expensive crowd member. Two different scheduling **strategies** have been implemented and evaluated. The first one, a random strategy, picks from the set of available and capable crowd members randomly the next candidate for the assignment. The second one, the metrics assisted strategy, uses the previously presented metrics in Table I to select the next candidate from the set. During an independent task assignment the metrics weight factor is equally distributed. Nevertheless, this strategy is also aware of task dependencies. Once a parent task is assigned, an extra weight (60%) is set to the *overlap* metric to move the most overlapping members to the begin of the selection queue.

### B. Experiment Results

The main goal of the experiments is to illustrate the effectiveness of the metrics enhanced scheduling algorithm outlined in Algorithm 1; also compared to random scheduling. In the first set of experiments Figures 4(a), 4(b), 4(c), and 4(d), the percentage of *completed tasks* with respect to the total number of assigned tasks is an indicator for the effectiveness. In the next set of figures (4(e), 4(f), 4(g), and 4(h)), the *exceeding costs* are considered. This second type of experiments illustrates the percentage of completed tasks that exceed the SLA cost objective. As aforementioned, this is caused by members that exceed the cost with their fee. The results of the two sets are presented by the rows in Figure 4. Furthermore, to explore lower, and in particular, upper bounds of the algorithm different batch sizes of tasks are scheduled. A batch size defines the number of tasks that are received from the EP and need to be scheduled in the next period. In our simulations, once all tasks of a batch assignment are past their deadline, a new similar size batch




(a) Processed - 256 batch. (b) Processed - 512 batch. (c) Processed - 768 batch. (d) Processed - 1024 batch.

(e) Over cost - 256 batch. (f) Over cost - 512 batch. (g) Over cost - 768 batch. (h) Over cost - 1024 batch.

Figure 4. Results of the experiments for various scheduling strategies with different task batch size. Strategies include *random* scheduling with *no* rescheduling phase *(rn)* and with rescheduling *(r)*, according to metrics *ordered* scheduling with *no* rescheduling (on) and with rescheduling (o), and finally, metric-ordered scheduling including *costs* with *no* rescheduling *(cn)* and with rescheduling *(c)*.

is scheduled until a number of 10000 tasks total are assigned. Note, the fixed size crowd has a maximum capacity of 768 tasks per period. The columns of the results in Figure 4 represent the different batch sizes.

As one immediately realizes form the first row of results, the amount of successfully processed tasks decreases with increasing batch size. From left to right, the bar at the very left of the experiment figures shows the results for a random scheduling strategy with no rescheduling phase **(rn)**. Together with the next bar, representing random scheduling with rescheduling **(r)**, they perform the worst with task processing peeks of only 40% for **(rn)**, and a few more than half (54%) for **(r)**, respectively, at a batch size of 256. Even with the rescheduling enabled, this strategy cannot be considered satisfactory for any of the batch sizes. An interesting fact is however, that their success rate remains the same for the 512 and 768 batches. This indicates that for this strategy a medium to high task queue load results in a similar success rate. The next two bars represent metric-ordered scheduling. In contrast to the 4th bar (**(o)**) the 3rd bar shows the result without rescheduling phase **(on)**. Starting with 97% and 99%, respectively, for **(on)** and **(o)** at size 256, they decrease to 55% and 67% at size 1024 when task queues are too small to schedule all tasks. Here the improvement of rescheduling phase is apparent when testing higher batch sizes. At size 1024, if a batch is to large for the task queues, the success rate decreases notably for both settings. The last two configurations, cost aware ordering with no rescheduling **(cn)** and with rescheduling **(c)**, also consider costs when ordering the candidate set for a task. Interestingly, the impact to the success rate in comparison to cost-unaware ordering is only marginal. This is the result of simultaneity between the metrics.

The second row of experiments reflects the percentage of successfully processed tasks that, however, exceeded the expected costs. The results show, that with strategies that do not consider costs, the amount of tasks exceeding costs is around 20% for all batch sizes (first four rows). Only if the cost metric is included, costs can be saved for the minor and medium batch sizes. With size 256 almost no costs accumulate with **(cn)** and **(c)**. At size 512 half the costs can be saved as opposed to the other strategies. Starting with batches of size 768, the results of the two cost considering strategies diverge notably and perform similarly unacceptable with respect to cost-unaware strategies. At size 1024, for example, **(cn)** results in 20% over cost and **(c)** in 22%. As a synthesis, it can be observed that metric-assisted scheduling, generally, preforms twice as well as random scheduling. This remains true as long as task queues can schedule all tasks. Rescheduling helps to increase the success of processing in each case. Whilst the success difference remains on average around 9% for random scheduling, in metric-assisted scheduling the difference depends on the batch sizes with extremes at size 256 with only 2% difference and 16% at size 768. If costs are also taken into account, both random and metric-assisted scheduling perform comparably. The success of an additional cost factor for metric-assisted scheduling depends on the batch size and is negligible for large batch sizes.

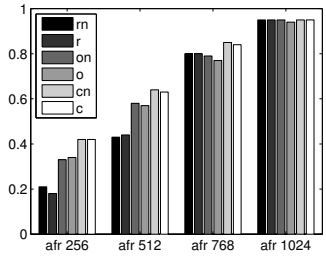



Figure 5. Assignment failure rate ($afr$) for different batch sizes.

The scheduling effort is another cost source of task assignments. Referring to the experiments in Figure 5 we list the assignment failure rate (*afr*) for the different batch sizes and varying strategies. The results show the percentage of scheduling requests that are rejected on the first request. The results highlight how the increasing batch size reduces the impact of the *afr* for the metric-assisted strategies (**(on)**,**(o)**, **(cn)**, and **(c)**). These strategies force similar member selections for comparable tasks. Thus, for small batches and cost on focus, the low fee members reject on full queues. For size 256 and 512 the effort difference between **(rn)**, **(r)** and **(cn)**, **(c)** is around 20% and decreases rapidly for larger sizes.

## VIII. Conclusion and Future Work

The main objectives of this work were to present a framework which successfully manages task assignment in a crowdcomputing environment. In this work we solve the problem with an adaptive and multi-objective task scheduling. Objectives derive from an agreement between a company and a crowd broker. As we base our environment on an SOA infrastructure with its convenient technical grounding for large-scale environments, we are also able to take advantage of the already existing models for agreements (WSLA and WS-Agreement). Our extension to one of the existing standards which includes assignment identifying information and relation to different objectives, fits the requirements of our crowdcomputing scenario. When deploying the assignment as independent and dependent tasks to capable members, these objectives can than be used as soft- or hard-constraints for a weighted scheduling strategy. The results of our experiments highlight the advantages of an objective-aware metric ordered strategy in contrast to plain random scheduling while task loads remain in between the boundaries. Nevertheless, the results show, the effort for ordering the assignment lists induces a higher effort in scheduling.

In future work we focus our research on two additional challenges. There remains the need for a fully automated translation of SLOs into the scheduling objectives. Vice-versa, this would also assist a semi-automatic approach for the negotiation phase where current policies are translated into SLOs that the crowd broker wishes to negotiate. Finally, a major challenge of crowdsourcing is the fluctuation and unpredictable behavior of the resources. As already mentioned in the scenario, however out of scope for this work, we see great potential in the role of the Head Hunter broker. Its main function is to lend the missing but required members to crowd brokers on resource shortage. For that purpose we plan to study methods and algorithms to derive tendencies of expertise evolution in crowdsourcing.

## Acknowledgments

This work is supported by the EU through the FP7 projects S-Cube (215483) and COIN (216256).

## References

[1] D. Brabham, "Crowdsourcing as a model for problem solving: An introduction and cases," *Convergence*, vol. 14, no. 1, p. 75, 2008.
[2] Amazon Mechnical Turk, http://www.mturk.com, May 2011.
[3] M. Vukovic, "Crowdsourcing for Enterprises," in *Proceedings of the 2009 Congress on Services*. IEEE, 2009, pp. 686–692.
[4] H. Psaier, L. Juszczyk, F. Skopik, D. Schall, and S. Dustdar, "Runtime Behavior Monitoring and Self-Adaptation in Service-Oriented Systems," in *SASO*. IEEE, 2010, pp. 164–174.
[5] H. Psaier, F. Skopik, D. Schall, and S. Dustdar, "Behavior Monitoring in Self-healing Service-oriented Systems," in *COMPSAC*. IEEE, 2010, pp. 357–366.
[6] LiveOps, https://www.livework.com/, May 2011.
[7] Yahoo! Answers, http://answers.yahoo.com/, May 2011.
[8] P. G. Ipeirotis, "Analyzing the Amazon Mechanical Turk Marketplace," *SSRN eLibrary*, vol. 17, no. 2, pp. 16–21, 2010.
[9] A. Andrieux *et al.*, "Web Services Agreement Specification." Open Grid Forum, 2007.
[10] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck, "Web Service Level Agreement (WSLA) Language Specification." IBM Corporation, 2003.
[11] H. Ludwig, T. Nakata, O. Wäldrich, P. Wieder, and W. Ziegler, "Reliable orchestration of resources using WS-Agreement," *HPCC*, pp. 753–762, 2006.
[12] Y. Che, "Design competition through multidimensional auctions," *The RAND Journal of Economics*, vol. 24, no. 4, pp. 668–680, 1993.
[13] D. Parkes and J. Kalagnanam, "Models for iterative multiattribute procurement auctions," *Management Science*, vol. 51, no. 3, pp. 435–451, 2005.
[14] J. Sousa, V. Poladian, D. Garlan, B. Schmerl, and M. Shaw, "Task-based adaptation for ubiquitous computing," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 36, no. 3, pp. 328 –340, May 2006.
[15] P. Cowling, N. Colledge, K. Dahal, and S. Remde, "The Trade Off Between Diversity and Quality for Multi-objective Workforce Scheduling," in *Evolutionary Computation in Combinatorial Optimization*, vol. 3906, 2006, pp. 13–24.
[16] A. Caprara, M. Monaci, and P. Toth, "Models and algorithms for a staff scheduling problem," *Math. Program.*, vol. 98, no. 1-3, pp. 445–476, 2003.
[17] A. T. Ernst, H. Jiang, M. Krishnamoorthy, and D. Sier, "Staff scheduling and rostering: A review of applications, methods and models," *European Journal of Operational Research*, vol. 153, no. 1, pp. 3 – 27, 2004, timetabling and Rostering.
[18] G. La Vecchia and A. Cisternino, "Collaborative Workforce, Business Process Crowdsourcing as an Alternative of BPO," in *Current Trends in Web Eng.*, vol. 6385, 2010, pp. 425–430.
[19] uTest, http://www.utest.com/, May 2011.
[20] D. Schall, H.-L. Truong, and S. Dustdar, "Unifying Human and Software Services in Web-Scale Collaborations," *IEEE Internet Computing*, vol. 12, no. 3, pp. 62–68, 2008.
[21] F. Skopik, D. Schall, and S. Dustdar, "Modeling and Mining of Dynamic Trust in Complex Service-oriented Systems," *Information Systems*, vol. 35, pp. 735–757, 2010.
[22] A. Agrawal *et al.*, "WS-BPEL Extension for People (BPEL4People), Version 1.0." 2007.
[23] J. J. Dujmović and H. L. Larsen, "Generalized conjunction/disjunction," *Int. J. Approx. Reasoning*, vol. 46, pp. 423–446, December 2007.