SmartOrch: An Adaptive Orchestration System for Human-Machine Collectives

Michael Rovatsos
University of Edinburgh, UK
mrovatso@inf.ed.ac.uk

Dimitrios I. Diochnos
University of Virginia, USA
diochnos@virginia.edu

Zhenyu Wen
University of Edinburgh, UK
zwen@inf.ed.ac.uk

Sofia Ceppi
University of Edinburgh, UK
sceppi@inf.ed.ac.uk

Pavlos Andreadis
University of Edinburgh, UK
p.andreadis@sms.ed.ac.uk

ABSTRACT

Web-based collaborative systems, where most computation is performed by human collectives, have distinctly different requirements from traditional workflow orchestration systems, as humans have to be mobilised to perform computations and the system has to adapt to their collective behaviour at runtime. In this paper, we present a social orchestration system called SmartOrch, which has been designed specifically for collective adaptive systems in which human participation is at the core of the overall distributed computation. SmartOrch provides a flexible and customisable workflow composition framework that has multi-level optimisation capabilities. These features allow us to manage the uncertainty that collective adaptive systems need to deal with in a principled way.

We demonstrate the benefits of SmartOrch with simulation experiments in a ridesharing domain. Our experiments show that SmartOrch is able to respond flexibly to variation in collective human behaviour, and to adapt to observed behaviour at different levels. This is accomplished by learning how to propose and route human-based tasks, how to allocate computational resources when managing these tasks, and how to adapt the overall interaction model of the platform based on past performance. By proposing novel, solid engineering principles for these kinds of systems, SmartOrch addresses shortcomings of previous work that mostly focused on application-specific, non-adaptive solutions.

Keywords

Distributed systems, workflow orchestration, workflow composition, workflow optimisation, collective adaptive systems

1. INTRODUCTION

In recent years, a new type of massive-scale distributed system has emerged in which most computation is performed by humans, e.g. social computing platforms such as Facebook and Twitter, collaborative content creation systems such as Wikipedia, crowdsourcing systems like Yahoo! Answers, human-based computation platforms such as Amazon Mechanical Turk, and sharing economy applications such as Uber and AirBnB. While specific programming frameworks have been proposed for such applications or to facilitate the use of existing platforms to enable a specific "social computation" [1, 3, 9, 10], the design of generic platforms to *orchestrate* human-centric collaborative computation has not received much attention in the literature.

Viewing *social orchestration* as the process of organising the interactions of *humans* with a computational platform and with each other, while at the same time managing the *computational* resources used by the system to support this organisation of activity, we obtain a set of requirements that are different from those of workflow management and enactment systems that are mostly "machine-centric" [2, 4, 19].

In order to identify these requirements, we need to observe the nature of operations performed by humans and machine *peers*, interacting over a digital (normally Web-based) platform. The overall *lifecycle* of performing a collaborative activity generally comprises (some or all of) the following stages: *peer discovery*, which identifies human peers that might be interested in contributing to a social computation, *task composition*, which generates possible (computational or physical) tasks these peers might perform individually or jointly, *task allocation*, which produces a concrete assignment of peers to activities (and may involve processes such as contracting or negotiation), *task execution*, which tracks the performance of these tasks, and *feedback*, which allows peers to report on their experiences, e.g. rate each other.

The *social orchestration* process managed by a platform deals with organising all this activity using computational means and through appropriate interaction with the users. It should be able to cater for differences in the specific requirements of each scenario while providing sufficient structure to support all stages of the above lifecycle. To conceptualise this, we distinguish between three *vertical* layers of computation involved in the *horizontal* lifecycle stages:

(1) The *process* layer, at which primitive computational processing steps are located (e.g. authentication, matchmaking). These cannot be broken down into separately orchestrated sub-steps, and need to be allocated computational resources per *platform job* that corresponds to a process instance to be executed. They can be performed by internal components of the platform, or delegated to external third-party services (e.g. a reputation service).

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

(2) The *orchestration* workflow layer, which embeds a concrete control flow of jobs enacted by a specific “manager” component, typically one for each of the stages of the lifecycle described above. For example, a task composition manager might obtain a request from a peer to generate suitable tasks to achieve a certain objective, and handle contacting the right peers, performing matchmaking operations, and consulting reputation information to rank the results.

(3) The *user interaction* workflow layer, finally, enacts the interaction model that governs the way users will work with the platform, providing the input/output interfaces to humans participating in the computation. For example, the negotiation process in a group activity might involve all participants in a task agreeing to it explicitly, and the process of tracking this agreement would be handled by a negotiation manager accepting agree/reject messages to update the status of the task. It is at this layer that managers expose APIs toward client applications, whereas the lower-level layers only involve APIs used either by platform components or internally used third-party services.

While this kind of horizontal and vertical composition of distributed processes has a lot in common with general distributed workflow orchestration systems, the paradigm of social computation brings new challenges with it that call for (1) more flexibility in terms of workflow composition and (2) adaptability required to be able to respond to observed behaviour. As pointed out in [3, 13, 17], once human activity needs to be orchestrated, it is difficult to predict patterns of use, quantity, timescales and quality of contributions at design time. This implies that we need to be able to flexibly extend and modify the structure of the system to respond to emergent collective human behaviour.

Consider, for example, the simple case of Web search, which provides a single-step, stateless user interaction workflow repeated millions of times every day. If users experience difficulty in finding what they are looking for, they will resort to using other applications, e.g. a crowdsourcing platform that utilises human input, whenever automated search does not produce good results. The disconnect between the two systems involved implies that the search engine provider does not receive feedback from users to modify the calculation of search results, or utilise crowdsourcing to complement them. Conversely, the crowdsourcing platform is unaware of the original queries posed to the search engine, and hence cannot mobilise its users at the time of the initial query. This simple example illustrates a much broader problem: Without the capability to compose more complex human-centric workflows and to adapt them to collective behaviour at runtime based on data observed across different parts of the overall environment, it is hard to manage such systems effectively.

A formal framework that allows general user interaction models, described as multi-agent protocols with different roles and activities, to be mapped onto orchestration architectures in ways that permit composition of complex application workflows has been previously described in [15]. Figure 1 shows an example of such a protocol, where a peer role p interacts with an orchestrator role o, traversing the different stages of the social computation lifecycle. In this *team task* protocol, peers can advertise capabilities C which the Task Composition Manager can use to generate a set of possible solutions T, i.e. tasks satisfying the description D contained in a task request. The participants of any proposed task $t \in T$ can agree or refuse to participate in it. Once all of them agree (a process tracked by the Negotiation Manager) they can start providing updates on the execution status of the task to the Task Execution Manager and feedback to the Reputation Manager. Note that any of these processing steps and interactions can be managed by human and/or machine peers. For example, updates on execution status could come from a sensor, or, conversely, suggestions of possible tasks could come from human users in response to a task request.

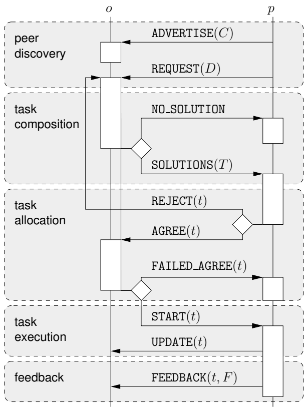

Figure 1: Team task protocol, adapted from [15]. Swimlanes represent peer roles, boxes their internal processing steps, and diamonds choice points.

The workflow composition capabilities of SmartOrch follows this model but are implemented using a single orchestrator to manage not just interactions with user clients contributing to different segments of the protocol at the same time, but also to run different protocols in parallel, reusing manager components where possible, and managing system resources in an integrated fashion.

In this paper, we present a novel social orchestration architecture called *SmartOrch* (*Smart* Architecture for Human-Centric Task *Orch*estration) based on above model. Its overall purpose is to manage a workflow, involving the performance of human and machine activities, to support the completion of domain *tasks* (e.g. getting a question answered, organising a meeting, renting a property), which may include steps that take place outside the platform in the real world. The contributions of these peers are enacted through API-level interactions with *data resources* exposed by the architecture, and whose state is handled by *managers*, each of which is a machine peer responsible for different parts of the workflow. The *orchestration manager* is a specific peer that allocates backend computational resources to the operations that need to be performed for the overall orchestration of such a workflow. It iterates over an asynchronous event processing loop, serving *platform jobs* maintained on dynamic process queues, one for every type of interaction with the system.

The key contribution of SmartOrch is that it enables *adaptation to collective human behaviour and computational performance* at all levels of the architecture through a uniform treatment of data collected by the system. At the level of individual *processes*, optimisation can be performed to take account of observed user behaviour in order to achieve the global design objective of the system, e.g. maximise uptake or resource sharing. At the *orchestration* level, the ways in which computational resources are allocated to different processes in terms of scheduling, parallelisation, and delegation, can be optimised based on observed human and machine performance. At the *interaction* level, different patterns that determine how the platform interacts with its users can be selected dynamically based on requirements and performance measures, and be deployed in parallel on the platform. At all three levels, information about past operations is tracked through provenance traces and computation profiling information. This information can be inspected and analysed by the maintainer of the platform, its users, and automated internal *optimiser* modules. This allows for flexible adaptation of an existing system configuration by humans and algorithmic procedures that perform automated adaptations. We believe that this multi-level adaptability is vital for collective adaptive systems, where overall performance largely depends on potentially volatile collective human behaviour.

The remainder of this paper is structured as follows: Section 2 describes the design of the SmartOrch architecture. Section 3 provides examples of adaptation at the process, orchestration, and user interaction level, and simulation experiments that demonstrate the impact of such optimisations in an example scenario. In section 4 we review related work, and section 5 concludes.

2. THE SMARTORCH ARCHITECTURE

SmartOrch has been developed as a purely event-driven, asynchronous framework that follows a fully RESTful design. At the level of backend processing, the orchestrator manages individual *platform jobs* by maintaining dynamic queues, one for every type of platform job. Jobs can be *created* due to a client-side interaction, they can be *triggered* as a side-effect of managers' activities (e.g. to synchronise different resources), or can be *delegated* to third-party services (e.g. for authentication). Events corresponding to steps in the user interaction workflows as perceived by the clients of the system are communicated to SmartOrch through REST APIs, as are internal events resulting from the operation of managers and other third-party services described below. SmartOrch responds to any such event by generating a sequence of jobs that need to be executed. Example jobs that occur as a result of almost all client-centric events are authentication, access control, and document validation. These sequences of jobs give rise to the *orchestration workflows* executed whenever calls to exposed APIs are received.

Developers can either use predefined jobs from a *job repository* to construct such workflows, or customise the response of SmartOrch further by writing custom jobs for a specific application. Distributed state is maintained through a set of exposed *resources* (documents), such as *task requests* that correspond to requests from users for a new task, *tasks* corresponding to composed tasks, and *task records* that are used to record execution updates. These documents are fully versioned, access-controlled, and linked to each other to reflect the lifecycle of their creation (essentially, each of them corresponds to a "message" arc in Figure 1 and contains the data corresponding to the content of the respective message).

In our implementation, peer profiles (used for peer discovery), reputation information (used in the feedback stage), and provenance tracking (used for optimisation, as explained in Section 3) are handled by services running on remote servers, while task composition, negotiation, and task execution management are handled by manager components running on the orchestration server. However, SmartOrch is indifferent to the physical distribution of these services.

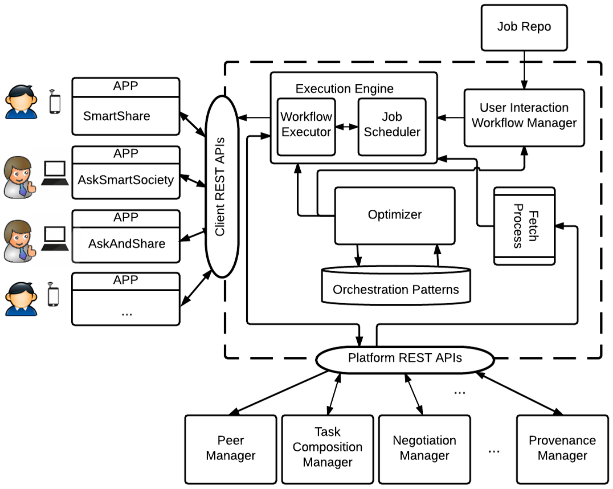

Figure 2: The Architecture of SmartOrch

2.1 Architecture structure

Figure 2 shows a high-level overview of SmartOrch and the main interactions between its components. *Core components* are shown inside the dashed box, and *supporting services* are shown at the bottom of the figure. Interactions between core components and client applications as well as supporting services are via *client* and *platform* REST APIs, where only the client interfaces are exposed for external use, whereas the platform APIs require privileges only provided to components used by the SmartOrch platform internally.

Creation of a specific orchestration system starts with the developer defining individual platform jobs from scratch or fetching existing implementations already published on a *Job Repository* and binding them so that different user interaction workflows can be created. Such jobs will typically be wrapped in orchestration workflows capturing a response to an API call. Consider the example of handling an `AGREE`(t) call during negotiation, which indicates that a peer p wants to join task t: An example orchestration workflow for this might involve authenticating p on the platform and checking whether p has write access to the document representing t, validating the format and contents of the data p is sending, recording p's agreement by updating t, and recording provenance information for this operation, which logs which peer (including machine peers) performed what operation on which data structure.

This particular workflow is handled by the Negotiation Manager, and contains calls to the Peer Manager (authentication, access control) and the Provenance Manager. If p is the final peer to agree to the task, the agreement will also trigger a call to the Task Execution Manager, as a new task record for t must be created. In other words, individual orchestration workflows are connected with each other to enact the user interaction workflow defined by client APIs.

The implementations of these orchestration workflows are stored in the *User Interaction Workflow Manager*, which is also used to select different workflows in different situations in the process of optimisation (cf. Section 3). This manager passes them on to the *Execution Engine*, which, in turn, contains the *Workflow Executor* and the *Scheduler*. The Workflow Executor receives orchestration workflows, dispatches the jobs they contain to the correct queues, and handles dependencies between them (for example locking a specific data resource while it is being updated). The Scheduler determines which pending job should be executed next by applying a prioritisation schedule on the different queues for jobs to be served. For example, simple document inspection jobs (e.g. when a user wants to display the list of current tasks on their client app) can be given higher priority than provenance recording, which is not time-critical, in order to ensure responsive behaviour of user interfaces.

The component that adds genuinely novel functionality to SmartOrch is the *Optimiser*, which extracts patterns from past operation of the platform that can be used to optimise the system at the process, orchestration, and user interaction workflow layers. To this end, it continually fetches provenance data and updates so-called *orchestration patterns* that describe what has been learned so far. The interactions between the Optimiser, the orchestration patterns, and other components of the architecture are further explained in detail in Section 3.1. The output of the Optimiser are decisions used as input for the User Interaction Workflow Manager and the Execution Engine.

The boxes at the bottom of Figure 2 represent manager components that support and use the orchestration backend, but whose internal structure is not controlled by this backend. Some of these, e.g. the Task Composition Manager, Negotiation Manager, and Task Execution Manager, are implemented using the SmartOrch Execution Engine, while others are third-party services interacting with SmartOrch only at API level. In the implementation of our example scenarios, these are: the Peer Manager, Provenance Manager, and Reputation Manager. The Peer Manager handles user registration and thus maintains profiles for human users, including credentials and scenario-specific profiles used for peer discovery. Any machine peers using platform APIs are also registered with the Peer Manager. The Provenance Manager we use is implemented using a remote PROV [12] server, and plays a key role in recording all operations at the level of accessing and manipulating data resources. It serves as the fundamental data source for the Optimiser's adaptation and optimisation functions, providing a uniform way of capturing observed system operation. Finally, the Reputation Manager is responsible for computing numerical reputation scores for each human user based on feedback supplied by participants in tasks that user was involved in. This can be used, for example, to rank the tasks proposed by the Task Composition Manager, or, as we will show below, to make decisions regarding which user interaction workflow to use for different sub-collectives of users in the system.

3. ADAPTATION CAPABILITIES

The most distinctive feature of SmartOrch is the way in which it embeds adaptation capabilities in the orchestration system. These enable it to respond to variations and changes in observed collective behaviour in order to ensure achievement of global system objectives and to optimise computational performance of the platform. While data-driven optimisation can in principle be performed in any orchestration architecture, by linking a component that has the capability to modify decisions at the processing, orchestration workflow, and user interaction workflow levels, and which operates on a single data source (provenance data describing all operations in the system) we enable a much more pervasive and principled capacity for runtime adaptation that is suitable for collective adaptive systems, where we expect variability in system behaviour to be the rule.

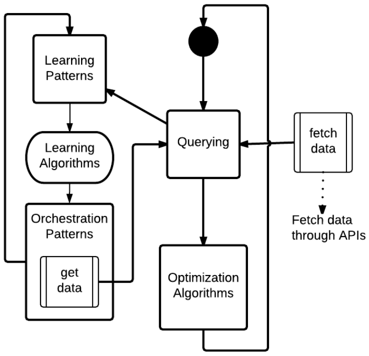

Figure 3: The SmartOrch Optimizer

3.1 Optimiser

The Optimiser follows the general design shown in Figure 3: Provenance data, describing all operations in terms of peers, resources, and activities performed on these resources, is continually gathered by the Provenance Manager and can be queried by the platform developer or an automated Optimiser process. For example, we might want to know how soon users stop using the platform after they have not found tasks that match their requests for a while. Optimisation algorithms (in a very broad sense of the term) make decisions based on the results of these queries, by using them as parameters that inform their decisions. In our example, if we find out that most users leave after three consecutive unsuccessful attempts, we should give those users priority over use of some resource (e.g. allocate them to a car in a ridesharing domain) after they have been unlucky twice. However, to discover such patterns, a background learning process must run in the background, which involves the use of *learning patterns*, defining what data patterns are queried to obtain learning data, and application of *learning algorithms* to extract pertinent *orchestration patterns*. In the above example, a suitable learning pattern might be

$$(\texttt{REQUEST} \rightarrow \texttt{NO\_SOLUTION})^n \rightarrow \texttt{REQUEST} \rightarrow \texttt{SOLUTION}$$

checking for all sequences where no solution was identified repeatedly n times, and then another request was posted by the same user later (using the notation of Figure 1). The orchestration pattern extracted from this might be

$$(\texttt{REQUEST} \rightarrow \texttt{NO\_SOLUTION})^3 \rightarrow \perp$$

where $\perp$ indicates no match against a future REQUEST. When these forms of background learning from data are applied, optimisation algorithms would query the library of extracted orchestration patterns, rather than the original provenance data directly.

It is worth noting that we use the notion of "orchestration pattern" very liberally here. For example, it could apply to a recurring pattern in a user interaction workflow as in the example above, to regularities in the performance of certain platform jobs (e.g. because of persistently high loads of specific services), and to many other similar cases. By and large, however, we are mostly interested in optimisations that aim at adapting to collective human behaviour.

3.2 Simulation Experiments

In this section, we present three different optimisations, one for each of the three layers of operation, performed by SmartOrch when it is used for a ridesharing application. The purpose of these experiments is to demonstrate the breadth of optimisation opportunities SmartOrch offers, and the benefits these can bring in terms of managing collective adaptive systems. The actual algorithms we use are deliberately kept simple, and do not constitute novel contributions.

A ridesharing application aims to support people that are willing to share a ride. In our simulation scenario, we consider users who want to go from location O to location D within the next 24 hours. Each of them is characterised by role (driver or commuter), gender (male or female), and age (young or adult). The aim of the system is to compute a global ride allocation, a solution that groups people into cars given their requirements and preferences.

The pickup location O, drop-off location D, and time of the ride are requirements of the users that impose hard constraints on the allocation proposed by the system. This information is elicited from the users during the peer discovery stage. During task composition, SmartOrch proposes only solutions to users that satisfy their requirements.

However, each user also has preferences about whom to travel with, i.e., they have *hedonic* preferences. We assume a setting in which young users want to travel only with other young users, adults only with adults, female users only with other females, and male users who are indifferent toward female and male users. This information is not directly elicited, so the system has to learn these hidden preferences and understand which type of implicit constraints they impose. In particular, to provide a solution that satisfies the users, the system needs to determine whether they would accept solutions that violate these preferences, i.e. whether they constitute hard or soft constraints in practice.

Assume that SmartOrch proposes a unique allocation $\mathcal{A}$ to users computed by a heuristic that maximises the sum of the utility of all the drivers, i.e., the drivers' social welfare (called "driver welfare" below). Formally, the utility $u(\mathcal{A}, d)$ of a driver d given allocation $\mathcal{A}$ is $u(\mathcal{A}, d) = k - k/|S(\mathcal{A}, d)|$ where k is the cost of the ride and $S(\mathcal{A}, d)$ is the set of users allocated in the car of driver d, assuming that the cost is spread equally among all passengers. The total welfare of all drivers D can be trivially computed as $U(\mathcal{A}) = \sum_{d \in D} u(\mathcal{A}, d)$. The heuristic the system uses is shown in Algorithm 1: In line 3, a set of teams T with one driver $d \in \mathcal{D}$ in each team is created. A team $\tau \in T$ represents a group of people that shares the same car in the solution proposed by the system. In line 4, given the capacity of the car of each driver $\mathcal{D}_i$, the system computes the total capacity C. The next steps identify the pair of commuter m and team τ that maximises the gain in the driver welfare by using the function `findmax`$(T, \mathcal{M})$ (Line 6), remove commuter m from the set of commuters that still need to be allocated (Line 7), and add commuter m to team τ (Line 8). These steps are repeated until no more commuters need to be allocated or the overall capacity C has been reached (Line 5). By randomly selecting the order with which users with identical requirements and preferences join a team, the algorithm can be proven to be fair in expectation. That is, in expectation, identical users achieve the same utility in the limit.

Algorithm 1 : Greedy task allocation algorithm

```
1: Input: a vector of drivers $\mathcal{D}$, a vector of commuters $\mathcal{M}$
2: Output: set of teams $T$ that compose allocation $\mathcal{A}$
3: initialize $(T, \mathcal{D})$
4: $C \leftarrow \sum_{i=1}^{|\mathcal{D}|}$ capacity $(\mathcal{D}_i)$
5: while $\mathcal{M} \neq 0$ and $\sum_{i=1}^{|\mathcal{D}|}(|T_i| - 1) < C$ do
6:     $\langle \tau, m \rangle \leftarrow$ findmax $(T, \mathcal{M})$
7:     $\mathcal{M} \leftarrow \mathcal{M} \setminus \{m\}$
8:     $T_\tau \leftarrow T_\tau \cup \{m\}$
9: end while
10: return $T$
```

Focusing on this greedy task allocation heuristic, we demonstrate the benefits of SmartOrch adaptation at the process, orchestration, and user interaction workflow levels.

Process layer. At the level of processes, we consider the example introduced in Section 3.1, whereby, after several iterations, the system learns that users drop out of the system when they are not matched to a car three times in a row. Given this, we introduce priorities in Algorithm 1 for users, which depend on how many times they have not been matched in a row. This results in a drastic reduction of the dropout rate: In a simple case with 100 commuters and 16 drivers offering 3 seats each, the user dropout probability is around 14% (i.e., $\left(\frac{100-3\times16}{100}\right)^3$) after three iterations with Algorithm 1, and 0 when user priorities are introduced. SmartOrch allows for this adaptation by modifying the job of computing tasks within the Composition Manager without reengineering the system. We simply store an additional attribute for every user in the Peer Manager that represents the number k of consecutive unsuccessful ride requests, and replace the `findmax` function by a procedure that sorts commuters by their priority, which increases with k.

Orchestration layer. To consider an example of adaptation at the orchestration level, we look at the performance of task composition in terms of driver welfare $U(\mathcal{A})$. Note that, given an allocation $\mathcal{A}$, we can compute the highest value of this quantity $U^*(\mathcal{A})$ that can be achieved when all users accept the proposed rides. However, we aim to use the actual welfare $U(\mathcal{A})$ achieved given actually accepted rides to evaluate the system, since this is a better measure of the quality of the solutions proposed. To evaluate user satisfaction we consider the two following measures: First, given an allocation $\mathcal{A}$, we consider the number of people $N(\mathcal{A})$ who accept rides and affect $U(\mathcal{A})$, compared to the maximum number of people $\mathcal{C}$ that could be sharing a ride (including drivers) in the best case, i.e., $\mathcal{C} = C + |D|$. Further, we also consider the difference between the number of users $N^*(\mathcal{A})$ to whom the system proposes a ride and the number of users that actually accept it, i.e., $N^*(\mathcal{A}) - N(\mathcal{A})$. Evaluating this difference is important because rejection of a ride can be interpreted as a failure of the system to understand user preferences. Finally, we measure computation time t of the algorithm to assess how much the efficiency of the system is improved. We compare the performance of SmartOrch with a system G that simply applies Algorithm 1. In what follows, we denote with $\mathcal{A}_S$ and $\mathcal{A}_G$ the allocations computed by SmartOrch and G, respectively.

In our example, the Optimiser knows that there are four subsets of users that should be considered separately in order to avoid ride rejections: young female users, adult female users, young male users, and adult male users. Given this, SmartOrch can dynamically learn to decompose the input set of the task composition problem into four non-overlapping subsets of users (who would not ride with each other anyway), and to run four instances of the algorithm in parallel, one for each subset of users, on a separate Composition Manager process. As a result, we expect to observe that $U(\mathcal{A}_S) \geq U(\mathcal{A}_G)$ even if $U^*(\mathcal{A}_S) \leq U^*(\mathcal{A}_G)$. Indeed, by discovering users' preferences and the type of constraints they impose, SmartOrch can avoid proposing rides that would be rejected. This guarantees that $U(\mathcal{A}_S) \geq U(\mathcal{A}_G)$. However, since no constraint related to preferences affects the allocation $\mathcal{A}_G$, $U^*(\mathcal{A}_G)$ corresponds to the upper bound of the driver welfare that can be achieved by any system, and thus, typically $U^*(\mathcal{A}_S) < U^*(\mathcal{A}_G)$.

Given the heuristic used in this example, the observations about the driver welfare trivially imply that $N(\mathcal{A}_S) \geq N(\mathcal{A}_G)$ and $N^*(\mathcal{A}_S) - N(\mathcal{A}_S) \leq N^*(\mathcal{A}_G) - N(\mathcal{A}_G)$. That is, users are more satisfied with solutions proposed by SmartOrch than with the ones proposed by system G because with the former system (i) more users are allocated and (ii) fewer users reject the solution proposed to them. Note that SmartOrch can flexibly decide how many instances of the algorithm to run in parallel and opt for the solution that, e.g., reduces the computational time of the algorithm without obtaining a driver welfare lower than that of G.

Before discussing the advantages of SmartOrch in terms of user interaction workflow, we present the results of the simulation of the example described above. The users considered in the simulation are randomly generated such that each of them is either male of female with 50% probability, young or adult with 50% probability, and are a driver with 20% probability. For the sake of simplicity, we assume all the users want to go from location O to location D during the same time interval. We consider different population sizes (50, 100, 500, 1000, 3000, 5000, and 10000 users) and the results we present are averaged over 100 instances for each population size.

First, we analyse the driver welfare shown in Figure 4 where $U(\mathcal{A}_G)$ and $U(\mathcal{A}_S)$ are compared to the upper bound $U^*(\mathcal{A}_G)$. The figure shows only results for populations up to 1000 users. It is easy to observe that the welfare achieved by SmartOrch is higher than the one achieved by G. We obtain similar results for every population size. Indeed, the average ratio between actual driver welfare and the upper bound, where the average is taken over all different population sizes, is 0.4777 for system G and 0.9829 for SmartOrch, with a standard deviation of 0.0037 and 0.0080 respectively. Note that, in this example, $U^*(\mathcal{A}_S) = U(\mathcal{A}_S)$ and thus $U(\mathcal{A}_S)$ in Figure 4 represents also the maximum welfare achievable when constraints due to preferences are taken into account. Thus, $U^*(\mathcal{A}_G) - U(\mathcal{A}_S)$ indicates how much preferences affect the maximum achievable welfare achievable. In our example, this effect is not significant.

We can make similar observations regarding user satisfaction. Figure 5 shows the number of users who accept the ride proposed by G and SmartOrch compared to $\mathcal{C}$, i.e. the maximum number of people that can take part in a ride. We observe that $N(\mathcal{A}_S) \geq N(\mathcal{A}_G)$, i.e. the users that accept the ride with allocation $\mathcal{A}_S$ are no less than the ones that accept it with $\mathcal{A}_G$. Moreover, more users reject the ride proposed by system G. In particular, $N^*(\mathcal{A}_G) - N(\mathcal{A}_G) > 0$ (light blue area), while $N^*(\mathcal{A}_S) = N(\mathcal{A}_S)$ (empty yellow area).

Finally, we discuss results for computation time of the algorithm shown in Table 1. Here, we compare the situation in which a system (like system G) always runs a single instance of the algorithm with that where execution of the algorithm is always parallelised whenever this reduces the expected number of rejected rides. When the system has 500 users or more, the parallel approach requires less time than the one used by G. This is not the case when there are no more than 100 users. However, since SmartOrch can dynamically adapt to specific situations, if it identifies that the computation time of the algorithm is more critical than, e.g., driver welfare, it can still decide to behave exactly as G, i.e., SmartOrch can decide how to run the algorithm by imposing $t_S = \min\{t_G, t_{parallel}\}$. This highlights the flexibility of SmartOrch in making decisions regarding trade-offs between performance depending on the situation in hand.

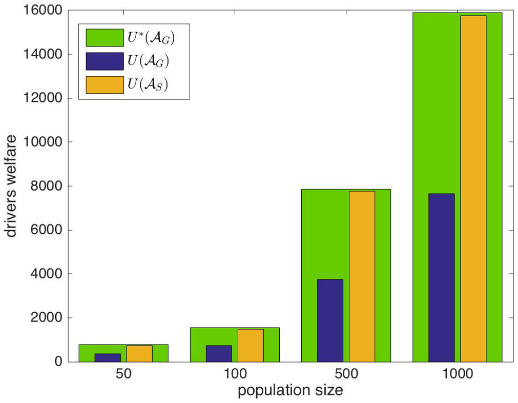

Figure 4: Drivers welfare

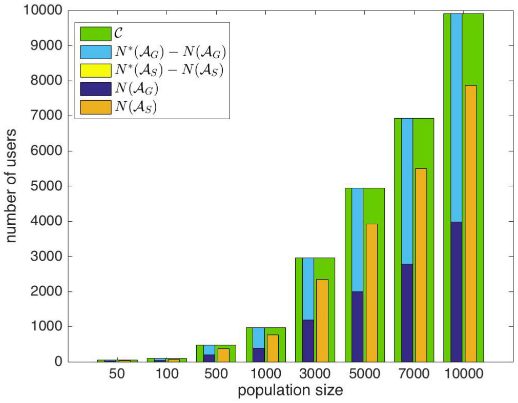

Figure 5: Number of allocated users

population size	t_G	$t_{parallel}$
50	$3.85 \cdot 10^2$	$18.82 \cdot 10^2$
100	$10.69 \cdot 10^2$	$20.74 \cdot 10^2$
500	$16.90 \cdot 10^3$	$3.49 \cdot 10^3$
1000	$65.18 \cdot 10^3$	$4.51 \cdot 10^3$
3000	$59.79 \cdot 10^4$	$1.14 \cdot 10^4$
5000	$157.75 \cdot 10^4$	$1.87 \cdot 10^4$
7000	$316.16 \cdot 10^4$	$2.59 \cdot 10^4$
10000	$636.87 \cdot 10^4$	$3.61 \cdot 10^4$

Table 1: Computation times (in seconds) of system G (t_G) and SmartOrch ($t_{parallel}$) for different population sizes

User interaction workflow layer. In the final set of simulation experiments, we focus on optimisations at the user interaction workflow level, specifically, when observing unreliable users. In the ridesharing scenario, in order to reach agreement, users have to accept a proposed ride in the task allocation stage. This requires a lot of explicit communication between the system and the users, and may lead to delays if users take some time to accept a ride. If we assume that each user i has a reputation r_i, we could re-

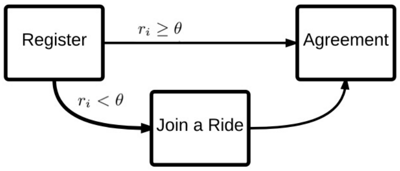

Figure 6: Workflow in which high-reputation users do not have accept rides explicitly

duce this communication overhead by selecting a threshold θ and requiring that only users with a reputation below θ must explicitly accept a ride. The idea is that users with a sufficiently high reputation are reliable enough to guarantee that it is unlikely that they will not accept a proposed ride.

Note that the potential loss in driver welfare this could incur is due to users who do not take the (explicitly accepted or not) ride proposed by the system, even if their requirements and preferences are satisfied, e.g. due to last-minute changes of plan that implicitly affects their reliability (and reputation). A user interaction workflow model that captures these two alternatives is shown in Figure 6.

We measure the impact of this more flexible workflow model on driver welfare as follows: We assign a reputation level $r_i \sim U(1, 100)$ to each user i. Since we assume that users requirements and preferences are satisfied, the probability p_i that a user i will not take the ride depends only on exogenous factors and thus on r_i. We model this probability as $p_i = (100-r_i)/100$, i.e., the complement of the reputation level. Moreover, we assume that if the user explicitly accepts a ride, her probability to "fail" to complete it correctly is reduced by 90%. For example, if $r_i = 20$, then $p_i = 80\%$ when i does not explicitly accept the ride and $p_i = 8\%$ when i accepts the ride. In our simulation, we use these probabilities to randomly select the users that do not take a proposed ride. For every population size and instance considered in the previous simulation, we present results averaged over 100 random selections. Driver welfare is computed by considering only users that actually perform a ride. Table 2 shows the proportion of driver welfare obtained compared to the maximally achievable value (achieved when $\theta = 100$ and every user has to explicitly accept the ride) for different threshold values.

size	$\theta = 0$	$\theta = 25$	$\theta = 50$	$\theta = 75$	$\theta = 100$
50	0.0438	0.4807	0.7500	0.9111	1
100	0.0444	0.4913	0.7598	0.9156	1
500	0.0472	0.5073	0.7753	0.9230	1
1000	0.0475	0.5088	0.7762	0.9233	1
3000	0.0476	0.5109	0.7778	0.9240	1

Table 2: Driver welfare for different population sizes and reputation threshold values

These results demonstrate the advantages of an adaptive system that performs optimisation at the user interaction workflow level, i.e. the flexibility of the system in deciding the threshold level depending on the particular situation it is currently exposed to. Indeed, SmartOrch can set the value of θ as required to trade off driver welfare against the number of messages exchanged in the system to orchestrate rides.

4. RELATED WORK

Recently, there has been much interest in systems that perform collective computation with significant human involvement, resulting of a growing body of research on human computation [8], social machines [6], human-agent collectives [7], and social collective intelligence [11]. Somewhat surprisingly, relatively little work within this space has focused on orchestration frameworks, let alone on optimisation and adaptation capabilities for such orchestration.

Programming frameworks for human computation such as TurKit [9], Jabberwocky [1], and AutoMan [3] provide high-level language support for "programming with people", using systems like Amazon Mechanical Turk or social networking platforms like Facebook as their backends. The proposed languages allow some flexibility, for example continuing the computation until a desired confidence level is achieved, but do not offer any facilities for composing complex multi-user workflows and optimising these over time.

CrowdLang [10] offers somewhat more extensive workflow composition functionality based on a number of generic collaborative patterns (e.g., iterative, contest, collection, divide-and-conquer), yet these are limited to crowdsourcing and do not aim to be used for other types of social computations. The same is true of the more comprehensive orchestration architecture proposed by Tranquillini *et al* [17], even though this allows for much more flexibility in the definition and enactment of custom task workflows.

Another line of work concerns specification languages for social computation-like systems. Murray-Rust *et al* [14] use a declarative process calculus for describing coordination protocols to capture collaborative processes among "social compute units", programmable abstractions of loosely coupled human teams. Their protocols are very similar to the models of user interaction workflows we presented in Section 1 (e.g. the one shown in Figure 1), and the experiments they conduct in a collaborative software development application scenario exhibit some adaptation dynamics in the ways in which the workflow is organised (the collective changes structure over time based on different events). However, their system is presented as a simulation prototype rather than as a reusable architecture, and the adaptations they describe are hardcoded into the system by the developer, not extracted from observed behavioural patterns.

Singh [16] presents "Local State Transfer" (LoST) as an architectural style for collaborative systems. He proposes a similar agent protocol-oriented description language for distributed process management and communication, which shares our focus on a data-centric, RESTful framework for communication and synchronisation. However, his framework does not map the formal enactment model onto a concrete computational orchestration architecture.

Beyond the area of social computation, there is of course a substantial body of work on general workflow management and service composition platforms [5], where the systems that share most characteristics with the kind of architectures required in our domains are those used for scientific collaboration such as Kepler [2], Triana [4], and Taverna [19]. Here, there is extensive work on adaptive workflow orchestration and optimisation, which, however, mostly focuses on scheduling jobs of a workflow over a set of resources.

For example, Wen *et al* [18] propose a method to partition scientific workflows over federated Clouds to meet security and reliability requirements, while minimising monetary cost. The authors of [20] introduce a method to transform the structure of a workflow such that the monetary cost of executing a workflow on a public cloud is minimised while providing performance guarantees.

These orchestration workflow optimisations are similar to some of the examples we present. However, they do not address the processing and user interaction workflow layers, the performance of which heavily depends on collective human behaviour, and has not been the focus of their systems.

5. CONCLUSIONS

In this paper, we have presented SmartOrch, a novel orchestration architecture for collective adaptive systems in which most computation is performed by human users.

The design of this architecture is heavily influenced by its human-centric focus in several ways: By implementing an asynchronous, event-driven orchestration loop, SmartOrch is able to cope with different user interaction workflows in parallel and with users engaging in different stages of the social computation lifecycle concurrently. These users will generally not exhibit the regularity of behaviour one might expect from computational processes. By applying a purely RESTful approach to managing state and shared data resources, SmartOrch exploits the standard architecture of the Web. This makes it easy to integrate different external services, build client apps for SmartOrch-based systems using simple, language- and platform-independent APIs, and exploit the scalability and robustness inherent to the infrastructure of the Web. By embedding adaptation and optimisation features at *all* levels of social orchestration, SmartOrch is capable of improving global performance based on continual observation of collective behaviour, both in terms of the computational characteristics of the system and the quality of social computations its orchestration activities result in.

To our knowledge, SmartOrch is the first orchestration system that combines these features and provides a framework generic enough to implement and combine a broad variety of social computation applications. Beyond the examples provided in the paper, SmartOrch has been used to implement further scenarios like meeting scheduling and chat-based group activity creation. We are currently working on extending its optimisation capabilities with domain-independent learning and resource allocation algorithms.

6. ACKNOWLEDGEMENTS

The research presented in this paper has been funded by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n. 600854 "*SmartSociety – Hybrid and Diversity-Aware Collective Adaptive Systems: Where people meet machines to build smarter societies*" (http://www.smart-society-project.eu/).

7. REFERENCES

[1] S. Ahmad, A. Battle, Z. Malkani, and S. Kamvar. The Jabberwocky Programming Environment for Structured Social Computing. In *Proceedings of UIST 2011*, pages 53–64. ACM, 2011.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *Proceedings of SSDBM 2004*, pages 423–, 2004.

[3] D. W. Barowy, C. Curtsinger, E. D. Berger, and A. McGregor. Automan: A Platform for Integrating Human-Based and Digital Computation. *ACM SIGPLAN Notices*, 47(10):639–654, 2012.

[4] D. Churches, G. Gombas, A. Harrison, et al. Programming scientific and distributed workflow with triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.

[5] E. Deelman, D. Gannon, M. Shields, and I. Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computing Systems*, 25(5):528–540, 2009.

[6] J. Hendler and T. Berners-Lee. From the Semantic Web to Social Machines: A Research Challenge for AI on the World Wide Web. *Artificial Intelligence*, 174(2):156 –161, 2010.

[7] N. R. Jennings, L. Moreau, D. Nicholson, S. Ramchurn, S. Roberts, T. Rodden, and A. Rogers. Human-Agent Collectives. *Communications of the ACM*, 57(12):80–88, Nov. 2014.

[8] E. Law and L. von Ahn. *Human Computation.* Morgan and Claypool, 2011.

[9] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. TurkIt: Tools for Iterative Tasks on Mechanical Turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, pages 29–30. ACM, 2009.

[10] P. Minder and A. Bernstein. Crowdlang: A programming language for the systematic exploration of human computation systems. In *Social Informatics*, pages 124–137. Springer, 2012.

[11] D. Miorandi, V. Maltese, M. Rovatsos, A. Nijholt, and J. Stewart, editors. *Social Collective Intelligence. Combining the Powers of Humans and Machines to Build a Smarter Society.* Springer-Verlag, 2014.

[12] L. Moreau and P. Groth. *Provenance: An Introduction to PROV.* Morgan and Claypool, 2013.

[13] D. Murray-Rust, O. Scekic, P. Papapanagiotou, H.-L. Truong, D. Robertson, and S. Dustdar. A collaboration model for community-based software development with social machines. *EAI Endorsed Transactions on Collaborative Computing*, 15(5), 2015.

[14] D. Murray-Rust, O. Scekic, H.-L. Truong, D. Robertson, and S. Dustdar. A collaboration model for community-based software development with social machines. In *Proceedings of CollaborateCom*, 2014.

[15] M. Rovatsos, D. Diochnos, and M. Craciun. Agent Protocols for Social Computation. In *Advances in Social Computing and Multiagent Systems*, volume 541 of *CCIS*, pages 94–111. Springer International, 2015.

[16] M. P. Singh. LoST: Local State Transfer - An Architectural Style for the Distributed Enactment of Business Protocols. In *ICWS*, pages 57–64, 2011.

[17] S. Tranquillini, F. Daniel, P. Kucherbaev, and F. Casati. Modeling, enacting, and integrating custom crowdsourcing processes. *ACM Transactions on the Web*, 9(2):7, 2015.

[18] Z. Wen, J. Cala, P. Watson, and A. Romanovsky. Cost effective, reliable and secure workflow deployment over federated clouds. *IEEE Transactions on Services Computing*, PP(99):1–1, 2016.

[19] K. Wolstencroft, R. Haines, D. Fellows, et al. The Taverna Workflow Suite: Designing and Executing Workflows of Web Services on the Desktop, Web or in the Cloud. *Nucleic acids research*, 4:W557-61, 2013.

[20] A. C. Zhou and B. He. Transformation-based monetary cost optimizations for workflows in the cloud. *IEEE Transactions on Cloud Computing*, 2(1):85–98, Jan 2014.