CSSWare: A Middleware for Scalable Mobile Crowd-Sourced Services

Ahmed Abdel Moamen and Nadeem Jamali(✉)

Department of Computer Science, University of Saskatchewan, Saskatoon, Canada
ama883@mail.usask.ca, jamali@cs.usask.ca

Abstract. The growing ubiquity of a variety of personal connected computational devices – each with a number of sensors – has created the opportunity for a wide range of crowd-sourced services. A busy professional could find a restaurant to go to for a quick lunch based on information available from smartphones of other people already there. Sensors on smartphones could detect whether their owners are having lunch, waiting to be seated, or even heading there.

Although the programming required for offering a new service of this sort can be significant if done from scratch, we identify core communication mechanisms underlying such services, which can be implemented as part of a middleware. Service designers can then launch novel services over this middleware by plugging in small pieces of service-specific code.

This paper describes the multi-origin communication mechanism which we believe to underlie many crowd-sourced services. It presents our design and prototype Actor-based implementation of middleware for crowd-sourced services, CSSWare. We present the code for a realistic crowd-sourced service to illustrate the ease with which new services can be specified and launched. Finally, we present our experimental results demonstrating scalability, performance and data-contributor side energy efficiency of the approach.

Keywords: Crowd-sourced · Middleware · Actors · Programmability · Power

1 Introduction

With the growing ubiquity of personal computational devices such as smartphones and wearable devices, has also come the ubiquity of sensors on these devices, as well as the potential for triggering actions virtually anywhere. This opens up an opportunity to offer a variety of services which rely on the state of the context in which devices are located, such as a person or a group of people carrying the devices, their geographical location, etc. We broadly refer to these as crowd-sourced services.

Consider a restaurant recommendation service which samples data collected about experiences of clients at a number of restaurants in a neighborhood and ranks them according to the service experience of these clients. The source of

S. Sigg et al. (Eds.): MobiCASE 2015, LNICST 162, pp. 181–199, 2015.
DOI: 10.1007/978-3-319-29003-4_11

the data could be sensor feeds on clients' smartphones, used to guess whether they are waiting, seated, enjoying their meals, paying or leaving. There could be a similar service for recommending hospital emergency services to people. Social media applications (Twitter, etc.) also appear to follow a similar pattern, where crowds contribute to collective messages by contributing free-form short messages.

We are interested in an opportunity created by the similarity in the patterns of communication required for many of these services, which we refer to as multi-origin communication. This is the type of communication where a group of senders contribute to a group message, without any of them necessarily *taking the lead*. Contrast this with a single-origin (multi-sender) communication [7], which is initiated by a single party which solicits interest from other parties to join together in sending a particular message. An example of the latter would be a workplace petition drafted by an individual and presented to others to sign. In multi-origin (implicitly also multi-sender) communication, the expectation is that there is no single party that takes the lead. In other words, multiple parties may autonomously launch messages which could then be aggregated in order to create a group message.

It turns out that unlike single-origin multi-sender messages, multi-origin messages require a setup in advance. Consider a public square where a number of citizens spontaneously begin to gather to party or protest. In this context, the physical space of the square serves as part of a setup which allows mutual observation, an opportunity to join in or leave, to endorse, reject or refine the collective message or experience over time. The closest electronic equivalent of such a physical space would be social media services such as Twitter, which allow people to observe others' tweets in an aggregate form (which is quite natural in physical space, but requires filtering and counting mechanisms in electronic space), endorse them by adopting hashtags, improve upon the message, and so on. In general, for a crowd (or mass) - conceived communication to happen, there is a need for a mechanism to be in place to coordinate the generation of the group message by soliciting individual messages, receiving them, and then aggregating them into a group message. The solicitation lays out the rules to be followed for selection of the potential senders, receiving their messages, and aggregating them. The communication could be one-time, periodic, or continual. There may or may not be a time-out for responding to the solicitation. All these aspects would be laid out in the original solicitation.

Multi-origin communication [1] serves as the core mechanism underlying many such crowd-sourced services. In other words, key coordination mechanisms can be provided in a platform over which a class of crowd-sourced services could be implemented relatively easily. Here, we present our efforts in realizing that potential by implementing a middleware for crowd-sourced services, CSSWare. Using CSSWare, all that a service designer needs to do to launch a new service is to identify a constituency of potential contributors, and to provide a few lines of service-specific code for specifying the nature of contributions and for aggregating them when they arrive. Additionally, we try to (opportunistically) optimize

the data contributor side energy consumption of crowd-sourced services for the situation where a number of services are being contributed to simultaneously. An optimizing sampling scheduler schedules the sampling of sensors based on the sensing requirements received from services running concurrently. The scheduler opportunistically optimizes the effective sampling rate of each sensor, exploiting opportunities for different services to share sensor samples when possible.

The rest of paper is organized as follows: Sect. 2 presents the related work. Section 3 describes our general approach to supporting crowd-sourced services using multi-origin communication. Sections 4 and 5 present our design and prototype implementation respectively. Section 6 evaluates the work by illustrating the ease with which new services can be implemented over our platform. It also presents our experimental results showing scalability, performance and energy efficiency of the approach. Finally, Sect. 7 concludes the paper.

2 Related Work

The term crowd-sourced can refer to two types of services: participatory sensing services and crowdsensing services. Participatory sensing involves explicit participation of human beings in possession of mobile devices, whereas crowdsensing relies on sensor feeds automatically flowing from devices to servers.

Participatory crowd sensing has been used in applications ranging from assisting drivers in making routing decisions based on real-time data (e.g., Waze [11] and TrafficPulse [12] to helping response to medical emergencies (e.g., CrowdHelp [5]) to disaster relief (e.g., in the aftermath of the 2010 Hatian earthquake [18]).

Among crowdsensing services, the real-time traffic information displayed on Google Maps is arguably the most widely used one, which now also has a participatory sensing aspect since Google's acquisition of Waze [11] in 2012. Uga et al. [15] have used crowdsensing to develop an earthquake warning system, which uses data from accelerometers present in many modern mobile devices to detect seismic vibrations.

Our work is more closely related to research focused on supporting crowd-sourced applications. Existing efforts have taken different approaches to supporting such applications in terms of programmability and generality.

Medusa [14] is a programming framework for crowd-sourced applications. A task (such as video documentation or citizen journalism) is launched by a requester, and *workers* are solicited through Amazon's Mechanical Turk (AMT) service. These workers – volunteering smartphone users – then provide raw or processed data to be used as part of a social or technical experiment. An XML-based programming language, MedScript, is used to specify the required task as a series of several stages, from the initial recruitment of volunteer workers, to the workers' (say, for a video documentation task) recording videos on their smartphones, summarizing them, and then sending them back. The stages can involve actions selectable from a library of executables, which are downloaded to mobile devices from a cloud server. Because Medusa requires that tasks pick from a limited set of activities, it suffers from limited programmability and generality, and is not applicable to a large class of crowd-sourced services.

AnonySense [6] is another framework for collecting and processing sensor data, which pays particular attention to privacy concerns. AnonySense allows a requester to launch one of a selected group of applications with their parameters. The application then distributes sensing tasks across anonymous participating mobile devices (referred to as carriers), and finally aggregates the reports received from the carriers. Achieving anonymity relies on separating sensor data from identifying features (such as homes or workplaces in GPS traces) to obscure individual identities. Similarly to Medusa, AnonySense has limitations in programmability and generality because of its limited focus on collection of sensor data and in-network processing.

CDAS [13] is an example of participatory crowd-sensing frameworks. It enables deployment of various crowd-sensing applications which require human involvement for simple verification tasks to deliver high accuracy services. Similar to CDAS, MOSDEN [10] is a collaborative mobile sensing framework that operates on smartphones to capture and share sensed data between multiple distributed applications and users.

The MECA (Mobile Edge Capture and Analysis) middleware for social sensing applications [16] focuses on efficient data collection from mobile devices. It uses a multi-layer architecture to take advantage of similarities in the data required for different applications to lower the demand on devices on which data is being collected. MECA's focus is limited to a narrow class of applications, and does not address wider programmability challenges. Furthermore, MECA – like other similar frameworks – uses the smartphone as a dumb data generator, offloading all processing to the server layer. This increases communication cost and does not allow applications to take advantage of data collected while the mobile device is not connected.

In summary, existing frameworks for crowd-sourced applications focus on narrow application areas or specific concerns, making it difficult to utilize them for a wider class of services. Also, none of them support concurrent execution of multiple services from within one service platform, which precludes taking advantage of opportunities to optimize for shared sensing requirements.

3 Supporting Crowd-Sourced Services

It turns out that a large class of crowd-sourced services exhibit a similar pattern of interaction, where members of a *crowd* contribute bits of information from their respective contexts, which are then aggregated to create useful information for clients. We have identified this pattern of interaction as *multi-origin (multi-sender) communication*, which involves aggregation of the messages received from a group of senders (referred here to as the constituency) into a *group message* to be sent on behalf of the group to one or more intended recipients.

Most examples of crowd-sourced services fit the *continual* type of multi-origin communication, where members of the constituency send messages on a continual basis rather than just once; this would be useful for a service provided over the web or through a mobile application where site visitors or application users seek

up-to-date information (say) on restaurant waiting times in a neighborhood. The *one-off* type of interaction soliciting only one message from each member of the constituency is a special case of this general case; this would be the type of communication used to serve one-time requests, such as to hold a census or an election, or to satisfy a one-off request to recommend a restaurant with a short waiting time. For some services, such as the one for restaurant recommendations, the choice between the continual and the one-off type of communication would depend on the frequency of requests, the number of potential senders of messages, etc. For instance, it would not be useful to be maintaining up-to-date information about all restaurants when there are very few requests for recommendations; however, it would be wasteful to solicit one-off communications for frequent requests.

From here on, we will refer to continual multi-origin communication as simply multi-origin communication.

3.1 Multi-origin Communication

To be precise in our presentation of continual multi-origin communication, we specify it in terms of the Actor model [3]. Actors are autonomous concurrently executing primitive agents (i.e., active objects) which communicate using asynchronous messages.[1] We represent the different parties involved in a multi-origin communication using actors, and define the required communication in terms of asynchronous actor messages.

The requester of a multi-origin communication makes a function call in order to launch the communication. The call passes two parameters, the first specifying the potential contributors – the constituency – to be invited to participate in the communication, and the second specifying an aggregation method. As illustrated in Fig. 1, an invocation of this function results in the creation of a new coordinator actor capable of coordinating the communication, which is next told to invite the constituency to participate. The coordinator then sends invitations to the members of the constituency (the contributors) to send their messages; when applicable, it also sends them parameters advising on how to construct their contributions (such as by tapping into a set of sensors, or soliciting input from the user), how often to send them (once or periodically, how frequently), etc.

As the contributors send their messages, the messages are aggregated by the coordinator as specified in its own behavior, to generate group messages on behalf of the contributors. When a contributor's message arrives at the coordinator, it checks whether the message warrants an update, or whether the interval for which it was to collect messages has passed. In both cases, it forwards an aggregate of messages received since the beginning of the interval to the requester. For example, a restaurant recommendation service available over the web would collect periodically sent updates from various restaurants and offer up-to-date information to site visitors.

[1] Actors are emerging as the model of choice for large-scale communication systems. Among others, Twitter and Facebook Chat have been implemented using Actor systems [4].

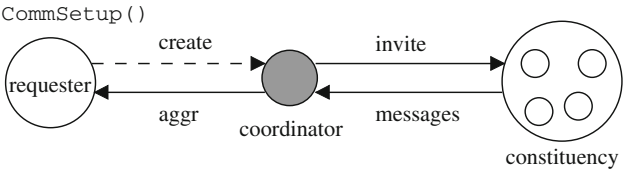

Fig. 1. Multi-origin communication setup

4 Middleware Design

Our design of a crowd-sourced service (CSSWare) middleware builds on the mechanism for multi-origin communication described in the previous section. As illustrated in Fig. 2, the sensing crowd becomes the constituency whose input is solicited. The service continually aggregates the feeds arriving from the crowd to create up-to-date custom views for various types of clients. For example, if the service were for recommending restaurants, one interface could be for prospective diners, another for the restaurant managers making real-time staffing plans, yet another could be for a vehicular routing system interested in improving downtown traffic flow at lunch time.

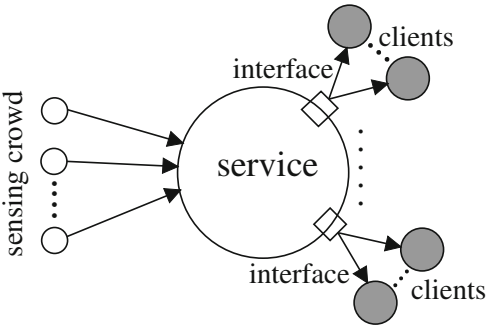

Fig. 2. Crowd-sourced service

Figure 3 illustrates how the distributed run-time system for the middleware is organized with parts executing on the service platform, on devices of members of the constituency, as well as client devices. In the rest of this section, we discuss these three parts separately.

4.1 Service Platform Side

The service designer uses the service creation API to create and launch a new crowd-sourced service. A set of parameters stating service specifications is passed through the API. These specifications identify the contributors to be invited

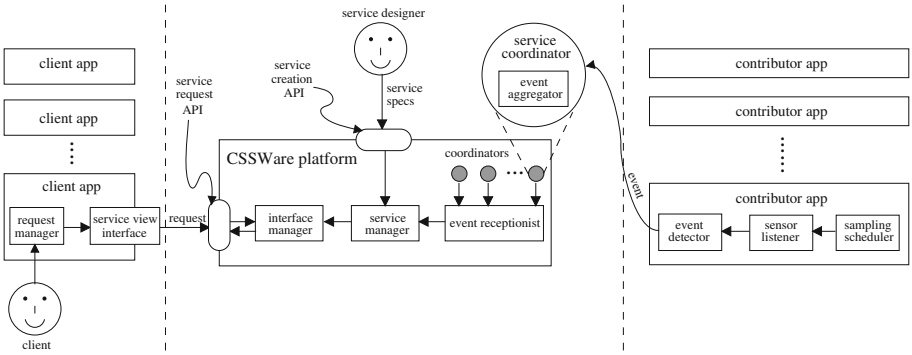

Fig. 3. System architecture

to participate in the service, the aggregation method to be used, as well as a description of the feeds solicited from the contributors in terms of specific events of interest, such as arrival at a restaurant, being seated at the table, etc.

To launch a new service, the service manager (see server in Fig. 3) creates a new service coordinator to coordinate the communication between the contributors and the CSSWare platform, which is capable of coordinating the communication between the contributors and the CSSWare platform. Next, it sends invitations to the contributors to send their events – when one is detected – to the coordinator. It also sends them parameters advising on how to detect events, construct their messages, and how often to send them (once or periodically, how frequently, etc.).

Contributor events received by a service coordinator are handled by its event aggregator, which in turn reports the events in aggregate form to the CSSWare platform's event receptionist. The aggregated events are then passed on to the service manager, which processes them to update the service's state, which is forwarded to the service interface manager to deliver appropriate views requested by clients through custom interfaces.

4.2 Contributor Side

To launch a service, the platform's service manager sends invitations to contributors to participate in the service. It also sends them parameters advising on how to detect events and construct their messages (i.e., sensing parameters). Event detection is carried out by dedicated *event detection actors*, who generate event feeds using relevant sensor feeds, which are then sent to the service coordinator.

An optimizing *sampling scheduler* schedules the sampling of each sensor based on the sensing requirements received from the service coordinator for each service being served at the time.

Sampling Scheduler
The scheduler attempts to optimize the sampling rate of each sensor exploiting opportunities for different services to share sensor samples when possible.

When the scheduler receives a new sampling request, it checks if the current sampling rate – sufficient for serving all currently served requests – can also satisfy the new sampling rate being requested; if so, it uses the existing sampling stream; otherwise, it changes the sampling rate to be high enough to accommodate the new request. The new sampling rate can be computed by finding the greatest common divisor of the existing and the newly requested sampling rates.

The sensor listener is responsible for sampling sensor data according to the sampling rate received from the sampling scheduler. However, because sensor samples are for all apps, there is a *filter* to extract the required samples to be sent to the different apps.

Algorithm 1 shows the steps followed by the scheduler to find the optimal sampling rate for sensing requests being served at the time. Each sensing request specifies the sensor to be sampled, as well as the rate at which it should be sampled. When a new request is received, the scheduler checks if the sensor is already scheduled; if so, it merges the current sampling rate with the GCD of the inverse of the current sampling rate and the new rate; otherwise, it sets up a new sensor listener to the requested sensor.

A more detailed presentation of the sampling scheduler can be found in [2].

Algorithm 1. Sampling Rate Adaptation Algorithm

```
1: procedure SENSOR SCHEDULING
2: Input: sensor name (s) and sampling rate (r)
3: Output: sensor data stream
4: /* check if s is already scheduled */
5: if SamplingScheduler.isSensorFound (s, r) is false then
6:     SamplingScheduler.add(s, r);
7:     create a new sensor listener actor for s
8: else /* if s is already scheduled */
9: /*find the GCD between r and current sampling rate*/
10:    newRate = GCD(SamplingScheduler.currentRate, r);
11: /* adapt the sampling rate */
12:    SamplingScheduler.adaptSamplingRate(s, newRate);
13: end if
14: filter sensor data
15: send sampling streams to services when the sensor listener detects an event
```

4.3 Client Side

A service can have various types of clients subscribed to different views of the service's state, each provided by a custom interface. When a client requests subscription to a particular type of view, the request manager inside the client

app constructs a custom view subscription request. This request is passed on to the service view interface, which is transmitted through the service request API of the CSSWare platform (see Fig. 3). The platform adds the client to a list of subscribers to that view of the service, and begins sending it all updates.

5 Middleware Implementation

A prototype of CSSWare has been implemented as an actor system. The prototype has two parts: a server implementing a crowd-sourced service platform (about 7,500 lines of code), and a mobile app supporting both client and contributor functionalities (about 4,600 lines of code).

Our implementation is built using the CyberOrgs [8] extension of Actor Architecture (AA) [9], a Java library and runtime system for distributed actor systems. Crowd-sourced services run over CSSWare, which runs over the CyberOrgs runtime system.

For the client and contributor side, we have ported AA to Android OS for supporting the mobile app.

5.1 Service Platform Side

To launch a new service, first, the requested service's meta data (i.e., its title and description) is added to the list of published services, which lists active services visible to contributors. Next, the service manager creates a service actor which invites potential contributors to send their events to the service's coordinator. It also sends them parameters advising on how to construct their contribution messages. After inviting the contributors, a new service view is created in the service request API in order to serve clients' requests.

As contributors to a service detect and send events, the events are aggregated by the coordinator and reported to the service manager through the event receptionist (see Fig. 3). The service manager collects aggregated events until a sufficient number of them have been received (as determined by a sufficiency condition provided by the service designer in the form of a function) and then updates the service state, revising the custom service views available to the clients.

5.2 Contributor Side

For the contributor (and client) side, we have ported CyberOrgs to Android OS, and implemented a self-contained application over it which runs on the Android OS (ver. 5.1). The current implementation supports contributions based on feeds from the GPS, accelerometer, microphone, magnetometer, gyroscope, pressure, humidity, temperature and light sensors. A set of high-level sensor events has been pre-implemented in terms of these (low-level) sensor events – as executable specifications – which a service designer can draw from and customize by providing parameters. These high-level events form the basis for

service events. For each high-level sensor event feed, the list of required low-level feeds is provided in the form of a list, where each entry identifies a sensor and specifies the rate at which it should be sampled. These specifications are typically only a few lines of code, varying between 7 and 18 lines of code for the triggers used in the example service prototypes. The code for using high-level sensor events to generate the service events is typically even shorter. The current prototype does not have a way for a service designer to add completely new high-level sensor or service event types; ongoing work is developing a way to allow that.

As shown in Fig. 4, the runtime system executing on the Android device has two components: the sampling scheduler and the event detector.

Sampling Scheduler. As described in Sect. 4.2, the sampling scheduler sets a sampling rate for each sensor based on the received sensing parameters. The scheduler first parses the service parameters to extract the coordinator name and the list of the service's event feeds.

Sensor listeners are responsible for sampling sensor data according to the sampling rate received from the sampling scheduler. The scheduler optimizes sensor sampling feeds by opportunistically sharing them between different service feeds.

Event Detector. Because the data sampled from a sensor can be for multiple event feeds, the data is filtered to extract the sub-feed pertinent to each event feed being served, and only that sub-feed is forwarded to the relevant event detection actor. An event detection actor monitors the sensor feed it receives for event triggers; when it sees one, it fires the event off to its service coordinator.

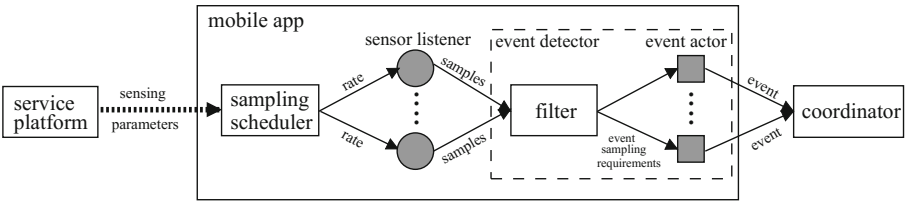

Fig. 4. Contributor side

An event detector does not maintain a local record of the triggered events itself; all events are sent to the service coordinator.

Because the contributor side of the system will likely execute on battery-operated mobile devices, it is important that contributors have the ability to either develop or adopt simple resource consumption policies to avoid undesired battery drain. We hope to utilize the fine-grained resource management features already present in the CyberOrgs [8] extension of Actor Architecture which we have used in our prototype. For now, we have implemented a feature allowing a service designer to specify resource limits after reaching which the contributor device would stop contributing feeds.

5.3 Client Side

Client side of the platform is implemented as part of the Android application implementing the contributor side. When a new service is launched, each client receives a notification about the launch. Multiple views are supported through custom interfaces installed by the service designer. A client interested in subscribing to a service can examine available views using the *service view interface* (see Fig. 3), and then use the service request API to subscribe to the desired view.

There is a collection of four general purpose view interfaces pre-implemented in the platform, which average at about 85 lines of code (the largest at about 100 and the smallest at 75 lines).[2] Although these interfaces are sufficient for the examples we have implemented, and for services with similar client side requirements, additional interfaces would need to implemented for different types of services. In our current prototype, there is no way for service designers to program these interfaces themselves; however, we plan to provide a way for new (general purpose or custom) interfaces developed by service designers or other parties to be installed or added to a repository from which they could be installed.

6 Evaluation

In this section, we present our evaluation of CSSWare for both the programmability of new services, as well as our experimental evaluation for performance, scalability and energy efficiency.

6.1 Programmability

The main programmability advantage of using CSSWare is in the orders of magnitude lower number of lines of code required for launching a new service. The prototype restaurant recommendation service presented in this section required 41 lines of code for the server and contributor side combined; in comparison, an equivalent standalone service we implemented required 6,142 lines of code. A twitter-like messaging service we implemented, which is not discussed in greater detail because of space constraints, similarly required 46 lines of code instead of 4,768 lines for an equivalent standalone service. For reference, the server and contributor end of the CSSWare platform required 7,473 and 4,622 lines of code respectively.

Restaurant Recommendation Service
Consider the type of restaurant recommendation service previously described in Sect. 1, where mobile devices of people visiting restaurants in a neighborhood automatically send real-time updates about the service they are receiving to a

[2] These 350 lines of code are included in the previously mentioned roughly 4600 lines of code for the Android application's implementation.

service provider, which then aggregates this information for people searching for restaurants. We assume that information required for generating these feeds can be gathered automatically by the devices by tapping into various sensors to determine when someone arrives at a restaurant, when they are waiting to be seated, when they sit down, when they are served, when they finish eating, and when they leave. The information could be coarser or finer grained depending on the device, usage habits, quality of the behavior detecting software, etc. These updates from personal mobile devices could then be aggregated by a service provider to rank restaurants according to criteria such as the amount of wait time before being seated, the length of time taken dining (shorter or longer, as preferred), the total amount of time that the user could expect to travel to the restaurant, dine, and be back at work. The ranking could also consider the server's meta-knowledge about the number of people being sent to various restaurants by the service.

Figure 5 presents our code implementing such a service as a `createSensorService()` method. First, a number of service variables are initialized: the list of restaurants (i.e., their names and coordinates), `restaurantList`, a method to be used by the coordinator to aggregate contributions, `aggrMethod`, and the sampling rate to be used for sensor feeds when a rate is not explicitly specified, `Default_SamplingRate`. `aggrMethod` is initialized here to a general purpose method for computing the average; it is to be used by the coordinator to compute average waiting time. Other services could use other available aggregation methods; our prototype provides a selection of them. There is currently no way for a service designer to add a new aggregation method, but we plan to provide that functionality in the future. Although here we hardcode the restaurants, functionality can be easily added to the mobile app to allow contributors to add previously unknown restaurants.

A sensor is set up for each of the sensor feeds required for any of the service feeds, following which the two types of service events are defined. The first, `locationEvent`, is defined to require the GPS sensor feed and is defined in terms of a number of parameters. The "trigger" parameters identify high-level sensor events, which become the basis for service events. For example `enterPlace` recognizes entering a location (a restaurant in this service). The "output" parameters identify the service events to be sent to the coordinator; here, `visitTime` computes the difference between `enterPlace` and `departPlace`. Additional parameter types are parameters that are available to the various methods; for example, `updateInterval` is available to `visitTime` as a parameter to decide the frequency of feeds to send to the coordinator.

Similarly, `activityEvent` specified a different sensor feed related to observations of the restaurant client's activity. It uses various sensor feeds. The triggers detect activities of "sitting down" or "being still," the latter using the `stillTime` parameter, which are then used as the basis for a `waitingTime` service event to be sent to the coordinator.

Finally, the service is created as an instance of the `CrowdService` class, and launched. The constructor for `CrowdService` takes as parameters a `title`, a

```
void createSensorService()
{
  /* initialize service variables */
  String placeList = "Restaurant1,52.1269,-106.7618;
           Restaurant2,52.1156,-106.5997; .......";
  int aggrMethod = ServiceEnum.average;
  int Default_SamplingRate =
     SensorManager.SENSOR_DELAY_NORMAL;

  /* defining sensors */
  Sensor GPS = new Sensor (ServiceEnum.GPS,
                           Default_SamplingRate);
  Sensor accelerometer = new Sensor(
  ServiceEnum.accelerometer, Default_SamplingRate);
  Sensor gyroscope = new Sensor(
      ServiceEnum.gyroscope, Default_SamplingRate);

  /* define a service event */
  ServiceEvent locationEvent = new ServiceEvent
  (ServiceEnum.sensorEvent, new List<Sensor>(){GPS},
    new List<EventParam>(){
      createParam("trigger",ServiceEnum.enterPlace),
      createParam("trigger",ServiceEnum.departPlace),
      createParam("placeList",placeList),
      createParam("updateInterval",30),
      createParam("output",ServiceEnum.visitTime),
   });

  /* define a service event */
  ServiceEvent activityEvent = new ServiceEvent
  (ServiceEnum.sensorEvent, new List<Sensor>(){
                      accelerometer,gyroscope},
    new List<EventParam>(){
      createParam("trigger",ServiceEnum.sitDown),
      createParam("trigger",ServiceEnum.still),
      createParam("stillTime",1),
      createParam("output",ServiceEnum.waitingTime),
  });

  /* create and launch the service */
  CrowdService service = new CrowdService(title,
           description, new List<ServiceEvent>()
    {locationEvent, activityEvent}, aggrMethod);
  service.launch();
}
```

Fig. 5. Restaurant recommendation service

`description`, the list of events (i.e., `locationEvent` and `activityEvent`) and the aggregation method `aggrMethod`. Once the service has been created, `launch` is called to launch the service.

6.2 Experimental Evaluation

We experimentally evaluated CSSWare in terms of performance, scalability and energy efficiency. Our experiments were conducted on a prototype Actor-based implementation of CSSWare. On the contributor side, we used a Samsung Galaxy Note II phone with a 1.6 GHz quad-core processor and 2 GB of RAM running Android OS ver 5.1. The server ran on a Windows 7 laptop equipped with a 2.6 GHz quad-core Intel i7 processor and 8 GB of RAM.

We installed instrumentation in the server and mobile application (i.e., contributor and client) parts of our prototype restaurant recommendation service to measure the processor time taken to perform various tasks. Instrumentation was also added to the contributor side to measure energy consumption of sensing.

Performance and Scalability

Service Platform Processing Demand. To evaluate the scalability of the server, we measured the resources required to host a service.

We created and launched a set of instances of the previously described restaurant recommendation service with their required frequencies of event feeds distributed over a normal distribution function. Specifically, we picked 150 random values with an average of 6.7, which added up to 1,000. We created 150 services with the randomly chosen feed frequency requirements, adding up to a cumulative feed frequency of 1,000 feeds per second. Each service received feeds from 10 restaurants. Note that the event feeds here are feeds of higher level events detected at the contributor end; these are not the raw data received at a high frequency from the sensors. In other words, the average frequency of 6.7 events per second per service would mean that something interesting is observed at some contributor device related to the service at the rate of 6.7 per second. Furthermore, we used a window size of 20 for recently received feeds for any window, this is the number of recent feeds which were used to compute a score for the restaurant. For this local aggregation, we simply maintained the average wait time for the restaurant, which required O(1) amount of time to maintain. These local aggregates for restaurants fed into the creation of a global aggregate in the form of a ranked list of the restaurants based on their scores, which amounted to a single step of *insertion sort* to maintain a sorted list, with an O(n) cost.[3]

Table 1 separately shows the one-time processing costs involved in creation of a new service as well as on-going processing costs as each event feed is received and processed. Creating service and coordinator actors – the former also including parsing the service's meta data (i.e., title and description) and adding the new service to the published service list – took 13.04 ms and 11.67 ms on average,

[3] Although this performs well for the small number of restaurants, it would be more efficient to use a binary search tree to keep a large number of restaurants sorted.

respectively. Initializing the global view for the service required 7.84 ms. In terms of on-going costs, receiving and parsing an incoming event feed required 7.35 ms on average. The cost of local aggregation to keep track of the average of the last 20 waiting times for a restaurant was 0.024 ms on average. This aggregation has O(1) complexity. We also measured costs for $O(\log n)$, $O(n)$ and $O(n^2)$ complexity local aggregation functions as shown in the table. The global aggregation for ranking the 10 restaurants incurred an average processing cost of 0.95 ms.

To put these numbers in some context, given the 8.325 ms required per feed on an on-going basis, about 120 event feeds could be processed by a server of our configuration per second. This could support a single service where 120 events are being collectively detected by the contributors every second, or 10 services which are each receiving about 12 feeds per second on average, and so on. In a broader context still, assuming 40 % of the population dines out at a meal time[4], assuming the diners are distributed somewhat evenly over a period of two hours, and each diner's device is sending 3 events over the course of their meal (indicating arrival, seating, departure) a server of our modest configuration could process 288,288 diners' data, equivalently data for a city of about 720,720 people. In practice, data from a small fraction of the diners could be used, allowing service for an order of magnitude higher population.

That said, our global aggregation function assumed only 10 restaurants. Although this may be reasonable because individuals requiring restaurant recommendations are not likely to be close to hundreds of restaurants, narrowing down the selection before aggregation would mean custom global aggregations, each costing the 0.95 ms. However, this custom aggregation could happen on the client's own device, without impacting the server's scalability. Alternatively, for a truly global aggregate for a city with (say) 10,000 restaurants, an $O(\log n)$ binary search tree could be used to keep the restaurants sorted; only the top few would ever need to be fetched, limiting the fetching cost.

Contributor Processing Demand. On the contributor side, again, we separately measured the initial cost of handling a new service's request for contribution, as well as the on-going cost of serving the service. The average total of measured one-time cost was 54.87 ms (SD 3.57). The on-going costs measured were per sensor feed: every time a piece of raw data was received from a server, its average total processing cost amounted to 8.68 ms (SD 1.02).

To put this on-going cost in perspective, about 115 sensor feeds per second could be handled on a device of our configuration (assuming no other computations executing). If an average service requires as many as 10 data samples per second (from a variety of sensors), 11.5 of such services could be supported; if an average of 1 data sample per second is required per service, a more likely scenario, 115 services could be simultaneously contributed to.

Client Processing Demand. For the client side as well, we measured the one-time processing costs of accessing a new service, as well as the on-going costs of receiv-

[4] Zagat 2014 restaurant survey reported that an average American ate out or bought 47 % of their lunches or dinners.

Table 1. Average processing time at the server side in ms

One-Time Per-Service Costs	**Mean**	**SD**
Create a service actor	13.04	2.63
Create a coordinator actor	11.67	1.74
Create a service view	7.84	0.98
Total processing time	32.55	5.35
Per-Event-Feed Costs	**Mean**	**SD**
Process an event feed	7.35	1.11
Local aggregation ($O(1)$ cost)	0.024	0.0021
Local aggregation ($O(\log n)$ cost)	0.078	0.0083
Local aggregation ($O(n)$ cost)	0.280	0.0349
Local aggregation ($O(n^2)$ cost)	0.680	0.0987
Global aggregation (10 Restaurants)	0.95	0.17
Total processing time ($O(1)$ local aggregation)	8.325	1.28

ing updates. The average total of measured one-time costs was 35.53 ms. The total of measured per-refresh on-going costs amounted to 60.9 ms on average, with 28.7 ms (SD 3.9) for processing the update, and 32.2 ms (SD 6.4) for display. In other words, a client could be simultaneously subscribed to and receive updates from 16 services every second. This is not very meaningful considering that more than half of the processing cost is for graphically displaying the update, which is not likely to happen simultaneously for more than only a few services. If we assume that only one service's updates are actually displayed at a time, more than 30 services could be supported in the background where interesting updates could lead to notifications, invitations to display, etc.

Energy Consumption of CSSWare vs. Standalone Services
Finally, a set of experiments was carried out to measure the overall improvement achieved in energy consumption by using CSSWare's sampling scheduler on the contributor device. We used the PowerTutor software [17] for our energy measurements.

To measure the overall improvement in energy consumption, we made measurements of energy used by CSSWare and identical standalone services implemented without using CSSWare. The sampling scheduler improved energy consumption of accelerometer and gyroscope sensors by up to 24.60 % and 26.63 %, respectively. However, the percentage savings depend entirely on the number of requests being served, because although the energy used is roughly linear in the cumulative sampling rate of all requests for the standalone services, for CSSWare, it depends almost entirely on the highest frequency being requested at the time, from which other requests are also served.

Overhead Analysis. In order to determine the non-sensing overhead of CSSWare, we measured the energy consumed by the contributor device side of the framework, albeit without the actual sensing. The average energy consumed was measured to be 72.9 mJ for the accelerometer, and a very similar 81 mJ for the gyroscope sensor. In percentage terms, this was roughly 4 % of the total energy consumed in the accelerometer experiments, and 0.8 % for the gyroscope sensor, the difference explained by the order-of-magnitude larger overall energy demand of the gyroscope sensor itself.

7 Conclusions

With the growing ubiquity of sensors and mobile devices, it is more possible than ever to offer innovative services based on both what the millions of sensors on people's devices are sensing, as well as what individuals are willing to actively contribute. However, the barriers to offering such services continue to be prohibitive for most: not only must these services be implemented, they would inevitably compete for resources on people's devices.

We have argued in this paper that many crowd-sourced services, including prominent social media services (if we consider their role of helping evolve collective messages), require similar communication mechanisms. We focus on one such mechanism – multi-origin communication – which allows a number of autonomous participants to contribute messages which can then be aggregated to create group messages on behalf of all. We introduced an approach to supporting crowd-sourced services using multi-origin communication, and presented our design and implementation of an Actor-based middleware for crowd-sourced services as a platform for launching such services. We illustrated the ease with which new services can be launched by presenting code for a prototype implementation for a crowd sensed restaurant recommendation service requiring fewer than 50 lines of main service specification code, with less than 100 lines of additional relevant code from available libraries of aggregation functions, feed specifications and service view interface. Finally, we experimentally evaluated the scalability of the approach. Most notably, even our modestly configured server could potentially provide a restaurant recommender service to a population of millions; contributor devices could contribute to tens if not hundreds of services simultaneously; client devices could monitor tens of services.

We have additionally addressed the challenge of satisfying the energy needs of a potentially large number of services requiring sensor data continuously. Use of the sampling scheduler takes advantage of the overlap in sensing requirements of various applications to achieve significant energy savings when there are overlapping requirements, with minimal overhead.

In on-going work, we are developing mechanisms for service designers and third parties to add new service feed specifications, custom service view interfaces, and aggregation functions. This will allow a larger variety of services to be implemented. We are also working on further simplifying programmability of services through web-based graphical interfaces. Finally, we would like to apply

our approach for fine-grained resource coordination to refining the sensor sampling scheduler, and more generally to manage the resource demands that a larger number of services may place on resource-constrained mobile devices.

Acknowledgments. Support from NSERC and CFI is gratefully acknowledged.

References

1. Abdel Moamen, A., Jamali, N.: Coordinating crowd-sourced services. In: Proceedings of IEEE Mobile Services, Alaska, pp. 92–99 (2014)
2. Abdel Moamen, A., Jamali, N.: ShareSens: an approach to optimizing energy consumption of continuous mobile sensing workloads. In: Proceedings of IEEE Mobile Services, NY, USA, pp. 89–96, June 2015
3. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
4. Agha, G.: Actors programming for the mobile cloud. In: Proceedings of ISPDC, pp. 3–9, June 2014
5. Besaleva, L., Weaver, A., CrowdHelp,: a crowdsourcing application for improving disaster management. In: Proceedings of GHTC, California, USA, pp. 185–190 (2013)
6. Cornelius, C., Kapadia, A., Kotz, D., Peebles, D., Shin, M., Triandopoulos, N., AnonySense: privacy-aware people-centric sensing. In: Proceedings of MobiSys, Breckenridge, USA, pp. 211–224 (2008)
7. Geng, H., Jamali, N.: Supporting many-to-many communication. In: Proceedings of AGERE!@SPLASH, Indiana, USA, pp. 81–86 (2013)
8. Jamali, N., Zhao, X.: Hierarchical resource usage coordination for large-scale multi-agent systems. In: Ishida, T., Gasser, L., Nakashima, H. (eds.) MMAS 2005. LNCS (LNAI), vol. 3446, pp. 40–54. Springer, Heidelberg (2005)
9. Jang, M.-W., Ahmed, A., Agha, G.: Efficient agent communication in multi-agent systems. In: Choren, R., Garcia, A., Lucena, C., Romanovsky, A. (eds.) SELMAS 2004. LNCS, vol. 3390, pp. 236–253. Springer, Heidelberg (2005)
10. Jayaraman, P., Perera, C., Georgakopoulos, D., Zaslavsky, A.: Efficient opportunistic sensing using mobile collaborative platform MOSDEN. In: Proceedings of the 2013 International Conference on Collaborative Computing: Observation of Strains: Networking, Applications and Worksharing (2011). Infect Dis Ther. 3(1), 35–43
11. Levine, U., Shinar, A., Shabtai, E., Shmuelevitz, Y.: Condition-based activation, shut-down and management of applications of mobile devices. United States Patents, US 8,271,057 (2009)
12. Li, R.-Y., Liang, S., Lee, D.-W., Byon, Y.-J.: TrafficPulse: a mobile gisystem for transportation. In: Proceedings of MobiGIS, California, USA, pp. 9–16 (2012)
13. Liu, X., Lu, M., Ooi, C., Shen, Y., Wu, S., Zhang, M.: CDAS: a crowdsourcing data analytics system. J. PVLDB **5**(10), 1040–1051 (2012)
14. Ra, M.-R., Liu, B., La-Porta, T., Govindan, R.: Medusa: a programming framework for crowd-sensing applications. In: Proceedings of MobiSys, pp. 337–350 (2012)
15. Uga, T., Nagaosa, T., Kawashima, D.: An emergency earthquake warning system using mobile terminals with a built-in accelerometer. In: Proceedings of the 2012 IEEE Conference on ITS Telecommunications (2012)

16. Ye, F., Ganti, R., Dimaghani, R., Grueneberg, K., Calo, S.: MECA: mobile edge capture and analysis middleware for social sensing applications. In: Proceedings of the Conference Companion on WWW, pp. 699–702. ACM, Lyon (2012)
17. Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R.P., Mao, Z.M., Yang, L.: Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In: Proceedings of CODES+ISSS, Arizona, USA, pp. 105–114 (2010)
18. Zook, M., Graham, M., Shelton, T., Gorman, S.: Volunteered geographic information and crowdsourcing disaster relief: a case study of the Haitian earthquake. World Med. Health Policy **2**(2), 7–33 (2010)