Coordinating Crowd-Sourced Services

Ahmed Abdel Moamen
Department of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada
moamen@agents.usask.ca

Nadeem Jamali
Department of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada
jamali@cs.usask.ca

***Abstract*—The growing ubiquity of smartphones and similar personal connected computational devices, each with a number of sensors, has created an opportunity for useful services based on crowd-sourced data. A busy professional could find a restaurant to go to for a quick lunch based on information available from smartphones of people already there having lunch, waiting to be seated, or even heading there; a government could conduct a census in real-time, or "sense" public opinion.**

Although the programming required for offering a new service of this sort can be significant if done from scratch, these applications have something in common: they use a similar pattern of coordinated communications between the various parties. This creates an opportunity to offer a set of coordination mechanisms as a platform to service designers, into which they can simply plug in their service specific code to offer a new service.

This paper identifies the coordination mechanisms required for these crowd-sourced services as types of multi-origin communication. We present details of how these core mechanisms can be implemented using Actors, and introduce high-level programming constructs for launching a new service. Finally, we use examples to illustrate the implementation of services.

I. Introduction

With the growing ubiquity of personal computational devices such as smart phones, Google glasses, etc., 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 group of people carrying the devices, their geographical location, etc. 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 the clients currently there. The source of the data could be sensor feeds on clients' mobile devices, which could guess whether they are waiting in line, seated, enjoying their meals, paying or leaving. As technologies like NFC payments become popular, more of these activities could be directly detected on the device rather than require guessing. The services themselves could also be less mundane. Consider, for example, a service combining real-time routing information – such as is collected for showing traffic information on Google Maps – with wait times at hospital emergencies to recommend which one to go to when in need of urgent care. If we also consider user input explicitly or implicitly entered into the devices, another class of services can be offered, from real-time polling, to instant censuses, and even voting in elections. The generation growing up in the world of Twitter and Facebook may find it quaint that democracies hold elections only once every few years. Why not have much more frequent votes on much finer grained decisions effecting citizens every day?

We are interested in an opportunity created by the similarity in the patterns of communication required for such services, which we refer to as multi-origin communication. In previous work [1], we have differentiated between single-origin and multi-origin types of multi-sender communication.[1] The single-origin type of multi-sender communication is initiated by a single party which solicits interest from other parties to join together in sending a particular message. An example of this would be a workplace petition. Using email, the option available is for one person to be the recognizable active sender of the petition, with the remaining people passively listed in the "cc" field. The alternative we developed allows all senders to be equally responsible for sending such a message, despite its single point of origin. In multi-origin (implicitly also multi-sender) communication, the expectation is that that there is no single party that must take 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

[1] We have previously referred to multi-sender communication as many-to-many communication; however, the emphasis was on the multiple senders, because there is sufficient existing research on multiple recipients. For multi-origin multi-sender messages, we skip multi-sender because it is implicit.

equivalent of such a physical space would be 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 message by soliciting 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. For example, imagine a multi-stage communication with the first solicitation being to invite nominations for topics to have the message on, followed by a vote to select the topic, followed by a solicitation of messages, followed by a final vote to agree on an aggregate message. 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 layed out in the original solicitation.

Our approach is to construct key coordination mechanisms required for this class of services requiring multi-origin communication, and then allow service designers to provide service specific code – which uses the available mechanisms – in order to launch their services. A new service could then be implemented by a service provider by simply providing the code for service-specific tasks, such as for solicitation of messages from mobile devices, the needed abstraction of sensor feeds on the devices, and aggregation of the messages to create a group message. These communications could be set up as continual real-time updates for a web or mobile app based service, serving user requests as they arise, or they could be launched each time a request comes in. The pieces of code provided by the service provider would simply be plugged into the common coordination mechanisms to create a new service.

The rest of paper is organized as follows: Section II presents the related work. Section III describes how to support multi-origin communication in crowd-sourced services. Section IV uses two examples to illustrate how services could be implemented using this approach. Finally, Section V concludes the paper.

II. Related Work

There have been a number of projects – both in academia and industry – involving crowd-sourced services. The term crowd-sourced can refer to two types of services: participatory sensing services and crowdsensing services. Participatory sensing involves explicit participation by the human being in possession of the mobile device, whereas crowdsensing relies on sensor feeds automatically flowing from devices to servers. We first present some representative examples of both these types of crowd-sourced services, and then discuss some existing frameworks for enabling such services.

A. Crowd-Sourced Services

Some of the best examples of participatory sensing services can be found in services aimed at assisting automobile drivers. Waze [2] is one of the largest community orientated mobile travel applications with users volunteering information about their driving experience in real time, by reporting on congestions, delays, and gasoline prices. These reports then become the basis for information displayed on other drivers' maps (on their mobile devices), to help them make routing decisions. Similarly, TrafficPulse [3] combines sensor data from mobile devices with real-time traveler reports from frequent travelers, and then offers this information to other drivers in an aggregated form.

Crowd-sourcing has also been found to be useful in efforts to coordinate rescue efforts following major disasters, such as the Haitian earthquake in 2010 [4]. Information aggregated from social media (e.g., blogs, emails, tweets, and facebook status updates) was used to overcome challenges created by both the inadequacy of maps and the change in landscape because of the devastation.

CrowdHelp [5] uses smartphones to collect direct feedback from mobile users about their medical condition, in combination with data coming from sensors in smartphones. This information is used to enable swift response to emergencies. For example, when CrowdHelp is used for emergency reporting, mobile users submit information relevant to an event (such as the number of injured people and their state) to a central server. This information is collected and sent to the nearest health care facility capable of treating the injured.

Among crowdsensing services, the real-time traffic information displayed on Google Maps is arguably the most widely used one. The service relies on location data voluntarily made available by users of Google's services, which is then aggregated and then visualized on Google's Maps to show traffic flow. Since Google's acquisition of Waze in 2012, Waze's participatory sensing service has now been combined with Google's crowdsensing service for providing real-time traffic flow information.

Crowdsensing has also been used by Uga et al. [6] in an earthquake warning system, which uses data from accelerometers present in many modern mobile devices to detect seismic vibrations. Devices send reports of likely seismic activity to a server which then aggregates the reports received to send out warnings.

B. Mobile Crowd-Sourced Frameworks

Medusa [7] 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. A task specified in MedScript – an XML-based programming language – typically specifies 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.

AnonySense [8] 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.

III. Supporting Multi-Origin Communication

As illustrated by Figure 1, multi-origin communication involves a number of autonomous senders sending messages which are somehow aggregated into a group message. However, as previously explained in the introduction, this type of communication requires an advance setup for coordinating the communication.

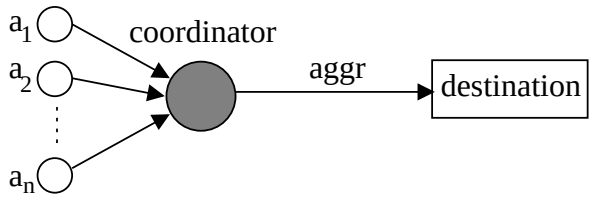

Figure 1. Multi-Origin Communication

We describe the implementation of this coordination setup as an Actor [9] program. Actors are autonomous concurrently executing primitive agents (i.e., active objects) which communicate using asynchronous messages.[2] We represent the different parties involved in a communication using actors, and define complex communications 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 parameters specifying the potential senders – the constituency – to be invited to participate in the communication, as well as the way in which the messages would be aggregated. Invocation of the 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 senders) to send their message; when applicable, it also sends them parameters advising on how to construct their messages (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.

We assume that each sender is an actor with a method to receive these requests, and the capability to create the types of messages. Given that there are a relatively small number of sensors on mobile devices, the parameters could simply be specifying which sensors to be tapping into, with what frequency, and what periods to be averaging the feeds over, etc. However, coordinators for some services may be more interested in hearing about higher-level events – such as a restaurant client sitting down at the table, finishing eating, paying the bill – which would require more significant local processing to generate than simply receiving sensor feeds. This could be supported in various ways: by migrating an actor with the required behavior to the sender, by sending the code as a parameter to create an actor locally, or simply by frequently updating the sender-side application to include the functionality needed by every type of request.

As the senders send their messages, the messages are aggregated by the coordinator according to its own behavior, to generate group messages on behalf of the senders.

We specifically introduce two types of such setups. The first – one-off multi-origin communication – is to solicit a group message from a number of senders with a termination condition and a timeout. 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. The second – continual multi-origin communication – is to solicit a continual feed of group messages from a number of senders. 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. For some services, such as the one for restaurant recommendations, the choice of one or the other setup 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.

A. One-Off Multi-Origin Communication

In a one-off multi-origin communication, the coordinator actor expects at most one message from any sender. It collects messages until either a sufficient number of messages has been received (as can be tested using a termination function), or a timeout has been reached; it then proceeds to aggregate the messages, and sends the aggregate to the requester on behalf of the senders. An example of a multi-

[2] Actors are emerging as the model of choice for large-scale messaging systems, most recently being adopted by Facebook's Messenger service.

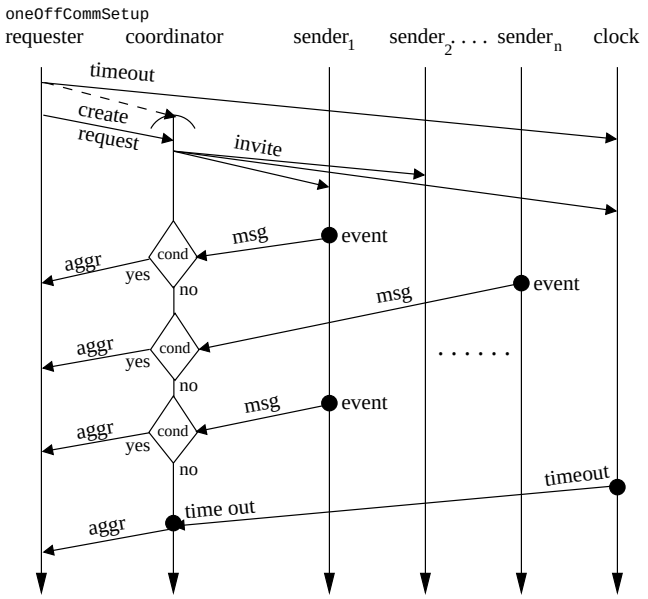

Figure 2. One-Off Multi-Origin Communication

origin communication with timeout would be an electronic voting service, where the coordinator expects no more than one vote from each voter and there is a deadline by which all votes must be in.

Figure 2 illustrates the execution of a one-off multi-origin communication using an actor event diagram [9]. In the figure, $sender_1$ through $sender_n$ are the prospective senders. There is a clock actor to which the requester sends a request to notify the coordinator when the timeout has been reached. We assume that the clock is local to the coordinator and has a way of notifying in a timely manner. The requester initiates the communication by calling the function `oneOffCommSetup(coordClass, constit, termCond, timeout)`, where `coordClass` is the desired behavior of the coordinator, `constit` is a list of senders, `termCond` is a function to test the termination condition indicating receipt of a sufficient number of messages, and `timeout` is a time when the coordinator would stop accepting messages from the senders.

Once the coordinator is created, it sends announcements to all senders, and begins collecting messages. The coordinator expects to receive the maximum of one message from each sender. After the timeout is reached, the coordinator sends a message to the requester with an aggregate of all responses.

The coordinator actor's behavior can be defined by extending the *multicall* selective blocking broadcast operation defined in [1] with support for timeouts, or directly using the following three methods:

- `announce(constit)`, used by the requester to instruct the coordinator to solicit messages from members of the constituency.
- `sendMessage(msg)`, used by the senders to send their messages to the coordinator.
- `timeout()`, used by the clock to tell the coordinator that the timeout has been reached.

```
void oneOffCommSetup(coordClass,constit,
                     termCond,timeout,custom) {
     coordinator = createCoordActor(coordClass,
                   termCond,custom);
     coordinator <- announce(constit);
     clock <- timeoutSetup(coordinator,timeout);
}
```

Figure 3. Pseudocode for oneOffCommSetup

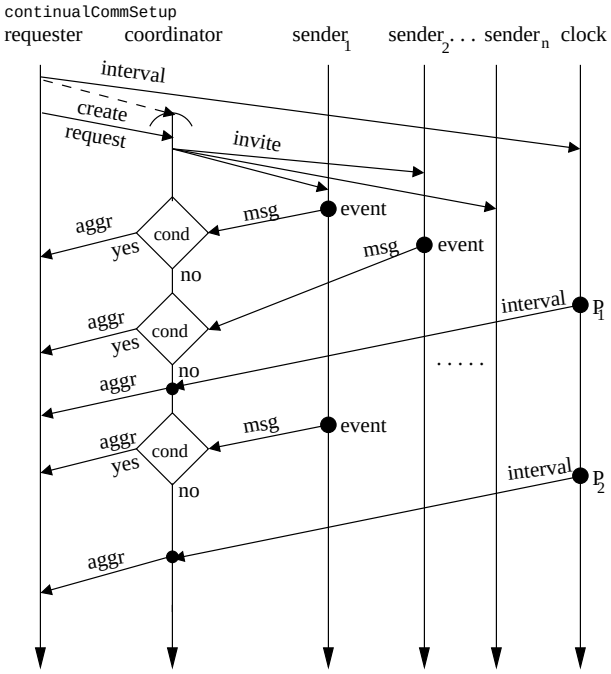

Figure 4. Continual Multi-Origin Communication

A sender actor's communication behavior is defined by one method: `receiveAnnouncement(serviceParams)`. This is the method invoked when the solicitation is received from the coordinator, and it carries out the computations specified in `serviceParams` in order to create its message.

Figure 3 shows pseudocode for the `oneOffCommSetup` function. The `create-CoordActor` function creates a new coordinator actor with the termination condition and application-specific customization initialized in its behavior, and returns the coordinator name. Once the coordinator has been created, a message is sent to the coordinator to broadcast an announcement to all senders. Another message is sent to the clock actor instructing it to notify the coordinator when the timeout is reached.

B. Continual Multi-Origin Communication

In a continual multi-origin communication, the coordinator expects multiple messages from each sender over time,

and periodically aggregates them and sends updates to the communication's requester. When a new message arrives, the coordinator checks whether it warrants an update, or whether the interval for which it was to collect messages has passed. In either case, it forwards an aggregate of messages received since the beginning of the interval to the requester. An example of continual communication would be that of a restaurant recommendation service available over the web, which attempts to offer up-to-day information to site visitors. The service could also be customized for individual visitors, based on their geographic locations, preferences, etc.

Figure 4 illustrates the execution of a continual multi-origin communication using an actor event diagram. $sender_1$ through $sender_n$ now send multiple messages over time, reporting local updates. Also, the clock actor periodically (i.e., after every `interval` period of time) notifies the coordinator of the passage of an interval, at which time the coordinator computes a new aggregate.

A continual communication is initiated by the requester by calling the function `continualCommSetup(coordClass,constit,updateCond,interval)`, where `coordClass` is the desired behavior of the coordinator, `constit` is the list of prospective senders, `updateCond` specifies the condition in which the requester should be immediately updated,[3] and `interval` specifies the intervals at which the coordinator would be notified by the clock.

Once the coordinator has been created, it broadcasts an announcement to all senders, and then waits to receive messages. Senders either send updates periodically or when they observe an interesting event (such as a change in the level of activity in a restaurant, for example).

A coordinator actor's behavior is defined by the following methods:

- `announce(constit)`, used by the requester to instruct the coordinator to solicit messages from members of the constituency.
- `sendMessage(msg)`, used by the senders to to send messages to the coordinator.
- `interval()`, used by the clock to inform the coordinator of the passage of each interval.

A sender actor's behavior is defined by one method: `receiveAnnouncement(serviceParams)`. This is the method invoked when the solicitation is received from the coordinator, and it carries out the computations specified in `service-Params` required for creating its messages.

Figure 5 shows the pseudocode for function `continualCommSetup`. The `create-CoordActor` function creates a new coordinator actor with an update condition and application-specific customization initialized in its behavior, and returns the coordinator name. Once

[3]This should also lead to resetting of the interval with the clock; this is not shown in the event diagram to avoid making it too crowded.

```
void continualCommSetup(coordClass,constit,
                        updateCond,interval,custom) {
     coordinator = createCoordActor(coordClass,
                   updateCond,custom);
     coordinator <- announce (constit);
     clock <- intervalSetup(coordinator,interval);
}
```

Figure 5. Pseudocode for continualCommSetup

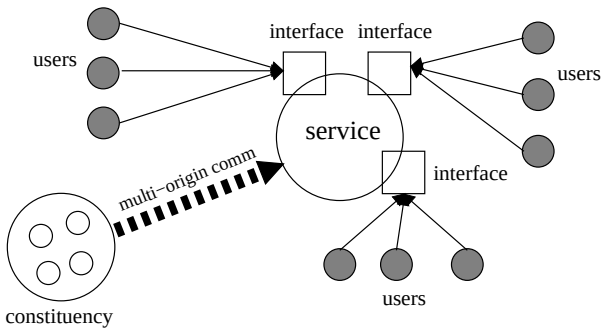

Figure 6. Design of Crowd-Sourced Services

the coordinator has been created, a message is sent to the coordinator[4] to broadcast an announcement to all senders. Another message is sent to the clock actor instructing it to notify the coordinator every time the required interval has passed.

IV. Usage Examples

Figure 6 illustrates how a crowd-sourced service could be designed using the multi-origin communication primitives we have described in the previous section. The service would

[4]$a \leftarrow m(p)$ means message m with parameters p is sent asynchronously to actor a.

```
void start() {
     * choose restaurants to track; assign them IDs;
       place them in restIDList with coordinates *
     for each restID in restIDList {
          * collect names of devices in or near restaurant ID *
          continualCommSetup(restCoordClass,
             deviceNameList,sigChange,null,restID);
     }
}

rankedRestList getView(location, rankParams) {
     return rank(filter(restIDList, location),
                                    rankParams);
}

void update(stateUpdate, restID) {
     * update global state with restID's new state *
}
```

Figure 7. Methods Defining Behavior of Restaurant Service Actor

solicit and receive one-off or continual multi-origin communications from a target constituency. These communications would enable the service to track the state of an activity, and various types of users would be able to request relevant views of the state through custom interfaces.

This section presents two case studies to illustrate the use of the two multi-origin communication primitives we have designed. Both examples use the continual multi-origin communication primitive; one-off versions of the examples can be easily adapted from these solutions using the one-off communication primitive.

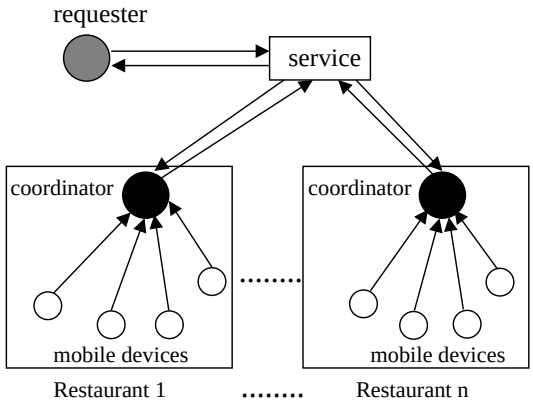

Figure 8. Restaurant Recommendation Service

A. A Restaurant Recommendation Service

Consider the type of restaurant recommendation service introduced earlier in the paper (Figure 8), where devices of people visiting restaurants in a neighborhood automatically send real-time updates about the service they are receiving to a service provider, who then aggregates this information for people searching for restaurants. We assume that information required for generating these real-time updates can be gathered automatically by a personal device (such as a smartphone) by tapping into various sensors to determine when some 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 behavior detecting software, etc. An aggregation of these updates could then be aggregated by the 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. The ranking could also include information about the number of people being sent to various restaurants by the service itself.

This service can be launched by creating and launching of a service actor, which in turn makes a number of calls to set up continual multi-origin messages, one for each

```
void sendMessage(deviceName,event,restID) {
      * record received event in eventList *
      if (sigChange(eventList))
           serviceName <- update(aggr(eventList),
                                 restID);
}
```

Figure 9. Methods Defining Behavior of a Regional Coordinator Actor

restaurant, each geographical area, etc., depending on the degree of distribution required or desired. The `start` method in Figure 7 shows how this could be done if a separate coordination were needed for each restaurant. The restaurants of interest are chosen, assigned unique IDs, and placed in a `restIDList`. Then for each `restID`, mobile devices in and near the restaurant are identified, say by tracking automatic *check-ins*. Finally, a call is made to set up a multi-origin communication primitive for each restaurant, with the nearby devices identified as the constituency. Additional parameters specify the condition indicating significant change in the restaurant state warranting an update to the server, and null to indicate that there is no set interval at which updates must be made. Each of these calls creates a local restaurant coordinator which invites event updates from current diners' devices. The devices in turn have applications installed to tap into sensor feeds to recognize significant events, such as arriving at the restaurant, being seated at a table. If there are a number of similar services that the device's owner is interested in, then each would interpret the sensor feeds for the purposes of that service. As an event gets recognized by a device, it sends a message to its restaurant coordinator, invoking the coordinator's `sendMessage` method (Figure 9). `sendMessage` records the event in `eventList` and checks to see whether the event represents a significant change in the restaurant's state, and if so, sends an `update` message to the restaurant service – known to the coordinator by its actor name `serviceName` – to report the change. Invocation of `update` in the service updates the global state with the new information. In a real system, it would also make sense for both the restaurant coordinators as well as the global service to use aging functions to lower the relevance of obsolete information.

A user searching for restaurants would call the `getView` method on the server with `location` and `rankParams` as parameters, where `location` specifies the user's geographical coordinates, and `rankParams` specify the metrics by which to rank the restaurant (such as by the wait time). The server filters the restaurant list for relevance according to the user's location, and then creates a ranking using `rankParams` to be returned to the client.

B. Twitter-like Messaging Service

Twitter serves a number of purposes, which include transmission of personal, organizational and news updates,

```
void createDiscussion(discussionTitle,constit) {
     * assign unique ID to discussionTitle *
     continualCommSetup(discCoordClass,constit,
               null,updateInterval,discussionID);
}

void getView(userName,userID,discussionID,
                    viewType,viewParams) {
     authenticate(userName,userID);
     userName <- view(filter(state,discussionID,
                userID,viewType,viewParams));
     * add userName's record to the subscriber list *
}

rankedMessageList findMessages(userName,
                    discussionID,keywords) {
     * create ranked list of existing messages relevant to keywords *
     return * ranked message list *;
}

void update(votesUpdate,discussionID) {
     * update state with votesUpdate *
     for each entry e in subscriber list {
          e.userName <- view(filter(state,
               e.discussionID,e.userID,
               e.viewType,e.viewParams));
     }
}
```

Figure 10. Methods Defining Behavior of Messaging Service Actor

```
void sendMessage(userName,userID,voteList) {
     authenticate(userName,userID);
     * record received votes in recentUpdates *
     if (currentTime >= lastAggregate + interval) {
         serviceName <- update(aggr(recentUpdates),
                                discussionID);
         lastAggregate += interval;
     }
}
```

Figure 11. Method Defining Behavior of a Discussion Coordinator Actor

networking, coordination of collective action, and sharing or propagation of opinions. Increasingly, it has also served as a source of information for journalists, opinion makers, politicians, etc. to acquire a sense of public sentiment. There are a handful of specific message formatting devices (particularly hashtags) which are created and subsequently adopted by contributors to indicate relationship with existing messages and conversations, and which enable some degree of analysis of sentiment. Here we discuss how to use the mechanisms we have presented in this paper to implement a service which allows users to both contribute their opinions, and obtain aggregate information helpful in assessing contributor sentiment.

Figure 12 shows how the service can be set up. The service is launched by the creation and launching of the messaging service actor, whose behavior is to receive requests for creation of new discussions with identified constituencies. These requests are received in the form of `createDiscussion` message sends as shown in Figure 10. When the service receives this message, it assigns a new ID – `discussionID` – to identify the discussion topic by, and calls the continual multi-origin communication setup primitive `continualCommSetup` with parameters specifying the discussion coordinator's behavior (`discCoordClass`), the `constit`, `null` for the update condition, `updateInterval` specifying the length of the intervals after which the service should receive updates from the coordinator, and finally `discussionID` to tell the coordinator its discussion topic ID. This call creates a dedicated discussion coordinator for that discussion, which in turn announces the discussion to the constituency. Once invited, members of the constituency are free to send messages to the discussion coordinator in the form of an asynchronous message invoking its `sendMessage` method (shown in Figure 11).

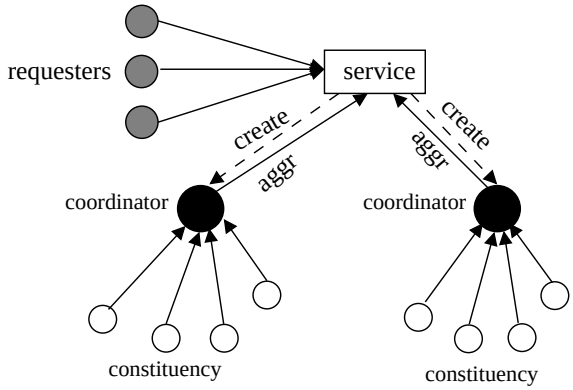

Figure 12. Twitter-like Messaging Service

`sendMessage` takes as parameter a list `voteList` of *(message, weight)* pairs, where *message* is either a new message drafted by the sender, or an existing message previously sent to the service (a ranked list of which can be obtained by calling the `findMessages` method of the messaging server), and *weight* indicates the proportional weight that the sender intends that message to have of their vote. Each sender has exactly one vote for any discussion, which they are free to distribute between various messages under their discussion.

The service can have various types of users, subscribed to different views of the discussions' states provided by custom interfaces (see Figure 12). When a user requests subscription to a particular type of view – `viewType` – after authentication, it is sent the view (by having a `view` message sent to it), and is also added to a subscriber list to be sent future updates. The types of view may include a view for an analyst interested in tracking trends, or even a view for a message sender interested in staying uptodate about a discussion to possibly revise their votes.

On receiving a `sendMessage` message, the discussion coordinator first updates `recentUpdates` to reflect the new messages received, and then checks to see if it is time to aggregate received messages and report back to the service. If it is time,[5] it aggregates the updates and reports them to the server using an `update` message, which invokes the corresponding method in the server. The server's `update` method updates the state of the discussion, and then for every entry in the list of service subscribers, sends them the view that they are subscribed to.

The service maintains the current state for all discussions. In practice, the service itself could be distributed into a number of actors, each handing any number of discussions.

There are some noteworthy features of this approach. First, message contributors are authenticated, and the voting is fair in that each contributor has the same one vote in any discussion, which they may divide among the multiple messages they support. Second, the constituency for each discussion is explicitly specified. This would allow this approach to be used for holding credible votes. Third, the approach naturally aggregates by allowing contributors to vote for existing messages rather than having them send a fresh message each time.

V. Conclusions

With the growing ubiquity of sensors, it is more possible than ever to offer services based on what the millions of sensors on people's mobile devices are sensing. However, the barriers to offering such services are significant. Not only must these services be implemented, they would inevitably compete for resources on people's devices. Our approach is to design a platform with key coordination mechanisms implemented to support a large class of services, and then allow service designers to focus their attentions on service specific code, which could simply be plugged into the platform. Not only would this speed up development of innovative services, the platform could also offer an opportunity to optimize the delivery of needed sensor feeds to competing services.

We view the pattern of communication required for such services as that of a multi-origin communication. Although there must still be one initiator – because such communications require an initial setup – the initiator's role is restricted to that of setting up a mechanism for coordinating the communication. The difference between this and single-origin multi-sender communication is subtle but important. Once the communication has been set up, within the constraints of the communication's design, the senders are free to contribute messages at their own pace, and the messages are aggregated to generate a collective message.

In this paper, we have described two simple but commonly occurring patterns of multi-origin communication, and have illustrated their use with examples. A prototype is currently in the process of being implemented over an Actor implementation also ported to the Android operating system to support crowdsensing. We are also looking at the patterns of communication in wireless sensor networks – which appear to broadly fit the criteria of multi-origin communication – to see if network routing approaches developed for WSN would also help optimize communication in our context.

[5] If messages are infrequent, a clock could be asked by the service to interrupt the coordinator at the end of each interval.

References

[1] H. Geng and N. Jamali, "Supporting many-to-many communication," in *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!@SPLASH)*. New York, NY, USA: ACM, 2013, pp. 81–86.

[2] U. Levine, A. Shinar, E. Shabtai, and Y. Shmuelevitz, "Condition-based activation, shut-down and management of applications of mobile devices," *United States Patents, US 8,271,057*, 2009.

[3] R.-Y. Li, S. Liang, D.-W. Lee, and Y.-J. Byon, "Trafficpulse: A mobile gisystem for transportation," in *Proceedings of the 2012 ACM SIGSPATIAL International Workshop on Mobile Geographic Information Systems*, ser. MobiGIS '12. New York, NY, USA: ACM, 2012, pp. 9–16.

[4] M. Zook, M. Graham, T. Shelton, and S. Gorman, "Volunteered geographic information and crowdsourcing disaster relief: A case study of the haitian earthquake," *World Medical and Health Policy*, vol. 2, no. 2, pp. 7–33, 2010.

[5] L. Besaleva and A. Weaver, "Crowdhelp: A crowdsourcing application for improving disaster management," in *Proceedings of the 2012 IEEE Conference on Global Humanitarian Technology*, ser. GHTC '13, 2013, pp. 185–190.

[6] T. Uga, T. Nagaosa, and D. Kawashima, "An emergency earthquake warning system using mobile terminals with a built-in accelerometer," in *Proceedings of the 2012 IEEE Conference on ITS Telecommunications*.

[7] M.-R. Ra, B. Liu, T. La-Porta, and R. Govindan, "Medusa: A programming framework for crowd-sensing applications," in *Proceedings of the 2012 International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 337–350.

[8] C. Cornelius, A. Kapadia, D. Kotz, D. Peebles, M. Shin, and N. Triandopoulos, "Anonysense: Privacy-aware people-centric sensing," in *Proceedings of the 2008 ACM International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '08. New York, NY, USA: ACM, 2008, pp. 211–224.

[9] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.