CSSWare: An Actor-Based Middleware for Mobile Crowd-Sourced Services

Ahmed Abdel Moamen and Nadeem Jamali
Department of Computer Science, University of Saskatchewan, Saskatoon, SK, Canada
ama883@mail.usask.ca, jamali@cs.usask.ca

ABSTRACT

The growing ubiquity of personal connected devices – each with a number of sensors – has created the opportunity for a wide range of crowd-sourced services. Although the programming required for offering new services can be significant if done from scratch, we identify core communication mechanisms underlying such services, and implement them in a middleware, CSSWare. Service designers can then launch novel services over CSSWare by simply plugging in small pieces of service-specific code. This paper describes the key multi-origin communication mechanism underlying many crowd-sourced services, presents our design and implementation of CSSWare, an actor-based middleware, and evaluates it programmability benefits and scalability.

Categories and Subject Descriptors

H.4 [**Ubiquitous and mobile computing**]: Miscellaneous

Keywords

CSSWare, Middleware, Crowd-Sourced Services, Actors

1. INTRODUCTION

With the growing ubiquity of personal computational devices has also come the ubiquity of sensors. This has opened up opportunities to offer services which rely on the context in which devices are located. Consider a restaurant recommendation service which samples data collected from customers at a number of restaurants and ranks them by wait times. In social media applications as well, crowds contribute to collective messages by contributing free-form short messages. We broadly refer to such services as crowd-sourced services.

We have previously [1] argued that the core mechanism underlying many crowd-sourced services is *multi-origin communication*, where a group of senders contributes to a group message, without any one of them taking the lead. Here, we use that approach to implement a middleware for crowd-sourced services, CSSWare, which allows services to be implemented using just a few lines of code.

Crowd-sourced can refer to participatory sensing services and crowdsensing services. Participatory sensing involves explicit participation by human beings in possession of mobile devices, whereas crowdsensing relies on sensor feeds automatically flowing from devices to servers. We discuss below some existing frameworks for enabling such services.

Medusa [5] is a programming framework for crowd-sourced applications, where tasks are carried out by volunteer *workers* who provide raw or processed data for social or technical experiments. Medusa has limited applicability because of the limited types of activities that the tasks can involve. MECA [6] is a middleware for 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 the devices on which data is being collected. MECA's focus is limited to a narrower class of applications, and does not address the wider programmability challenges which we do.

2. DESIGN AND IMPLEMENTATION

Most examples of crowd-sourced services fit the *continual* type of multi-origin communication, where members of a constituency send messages on a continual basis; this is useful for a service where clients seek up-to-date information (say) on restaurant waiting times in a neighborhood. From here on, we will refer to continual multi-origin communication as simply multi-origin communication.

We specify multi-origin communication in terms of the Actor model [2]. Actors are autonomous concurrently executing primitive agents (i.e., active objects) which communicate using asynchronous messages. The requester of a multi-origin communication makes a function call to launch the communication, with parameters specifying the potential contributors (the constituency) and the aggregation method. An invocation of this function creates a coordinator, which then invites the constituency to participate. As the contributors send their messages, the messages are aggregated by the coordinator, and forwarded to the requester.

Design. The design of our middleware builds on the mechanism for multi-origin communication. The sensing crowd becomes the constituency whose input is solicited (Figure 1). 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 for a vehicular routing system to improve downtown traffic flow at lunch time.

The distributed run-time system for the middleware is organized into three parts, executing on a platform server, on devices of data contributors and service clients. Figure 2 illustrates the organization of the platform server.

A service designer uses the service creation API to create and launch a new crowd-sourced service, passing parameters to identify the contributors, the aggregation method, and the types of feeds solicited from the contributors. The

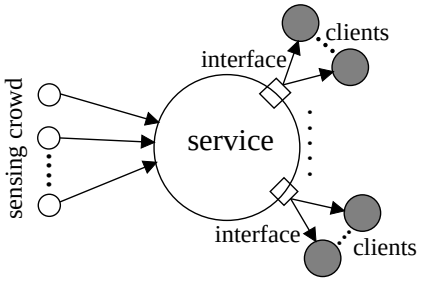

Figure 1: Crowd-Sourced Service

service manager then creates a new service coordinator and invites the contributors to send events to the coordinator. These events are aggregated by the coordinator's event aggregator and then reported to the platform's event receptionist. The service manager then uses these events to update the service's state, which is then transmitted to clients in their preferred views by the interface manager.

Implementation. The prototype has two parts: a server implementing the CSSWare service platform, and a mobile app supporting both client and contributor functionalities.

Our implementation is built using the CyberOrgs [3] extension of Actor Architecture (AA) [4], a Java library and runtime system for distributed actor systems. Crowd-sourced services run over CSSWare, which runs over CyberOrgs.

For the client and contributor side, we have ported CyberOrgs to Android OS, and implemented a self-contained application over it. Contributor events can use feeds from the GPS, accelerometer, microphone, magnetometer, gyroscope, pressure, humidity, temperature and light sensors. High-level sensor events has been pre-implemented in terms of these (low-level) sensor events – as executable specifications (of 7 to 18 lines of code) – which a service designer can draw from and customize by providing parameters. Service designers can also specify resource limits after reaching which the contributor device would stop participating.

A collection of four general purpose view interfaces is pre-implemented in the platform, which average at about 85 lines of code. We also plan to provide a way for new interfaces to be developed by service designers.

3. EVALUATION

We evaluated CSSWare for programmability and performance.

Programmability. CSSWare offers orders of magnitude smaller size of code for launching a new service. We implemented two prototype services – a restaurant recommendation service and a Twitter-like messaging service – as both services built over CSSWare and as standalone services.

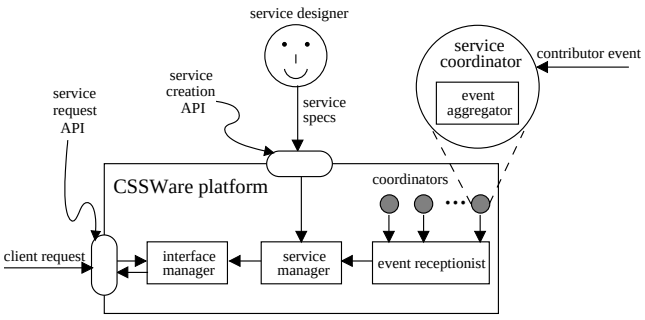

Figure 2: Platform Server

The restaurant and message services required 41 and 46 lines of code, respectively, for the server and contributor side combined. For comparison, the two services required 6,142 lines and 4,768 lines of code, respectively, when we implemented them as standalone services. For reference, the server and client/contributor end of the CSSWare middleware itself required 7,473 and 4,622 lines of code respectively.

Performance. We separately measured the processing costs of one-time service launching tasks and on-going tasks on both the CSSWare server, and the contributor/client side. The server ran on a Windows 7 laptop (2.6GHz quad-core i7, 8GB RAM), and the contributor/client app ran on a Samsung Note II (1.6GHz quad-core, 2GB RAM) running Android 4.4. The average one-time costs were 32.55ms (platform), 235.21ms (contributor) and 423.91ms (client), while the average on-going costs were 55.86ms (platform, assuming linear aggregation function), 293.89ms (contributor) and 675.4ms (client). For services requiring 1 feed per minute (second), this means support for 10,740 (179) and 2,040 (34) services on our server and mobile hardware respectively.

4. CONCLUSIONS

We argue that many crowd-sourced services, including prominent social media services, require similar communication mechanisms. We focus in particular on 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 introduce an approach to supporting crowd-sourced services using multi-origin communication, and present our design and implementation of an Actor-based middleware for crowd-sourced services as a platform for launching such services. We evaluate the approach by assessing programmability of new services in term of lines of code required, and provide experimental results on performance and scalability.

5. ACKNOWLEDGMENTS

NSERC and CFI support is gratefully acknowledged.

6. REFERENCES

[1] A. Abdel Moamen and N. Jamali. Coordinating crowd-sourced services. In *Proc. of Mobile Services*, pages 92–99, Alaska, 2014. IEEE.

[2] G. Agha. Actors: A model of concurrent computation in distributed systems. *MIT Press, MA, USA*, 1986.

[3] N. Jamali and X. Zhao. Hierarchical resource usage coordination for large-scale multi-agent systems. In *Massively Multi-Agent Systems I*, pages 40–54. Springer, 2005.

[4] M. Jang, A. Momen, and G. Agha. Efficient agent communication in multi-agent systems. In *Proc. of SELMAS*, pages 236–253, 2004.

[5] M.-R. Ra, B. Liu, T. La-Porta, and R. Govindan. Medusa: A programming framework for crowd-sensing applications. In *Proc. of MobiSys*, pages 337–350, 2012.

[6] F. Ye, R. Ganti, R. Dimaghani, K. Grueneberg, and S. Calo. MECA: Mobile Edge Capture and Analysis Middleware for Social Sensing Applications. In *Proc. of WWW*, pages 699–702, Lyon, France, 2012. ACM.