interActors: A Model for Separating Complex Communication Concerns in Multiagent Computations

(Extended Abstract)

Hongxing Geng*
Agents Laboratory
Department of Computer Science
University of Saskatchewan
hongxing.geng@usask.ca

Nadeem Jamali
Agents Laboratory
Department of Computer Science
University of Saskatchewan
jamali@cs.usask.ca

ABSTRACT

Multiagent computations increasingly require complex and diverse interactions. Existing approaches to separating these communication concerns offer static protocols, which cannot handle dynamically evolving numbers and sets of communicating agents, and leave complex initialization steps mixed in with functional concerns.

We present interActors, a model for separating complex communication concerns of applications from their functional concerns. These first-class communications are self-driven and dynamically change in response to evolving communication needs; they can be easily created, reused and composed. A prototype implementation is presented. A case study illustrates improved programmability using interActors.

1. INTRODUCTION

Concurrent multiagent systems increasingly involve complex and diverse interactions. Consider a finer-grained 21^{st} century version of democracy, where instead of electing politicians for years, citizens can contribute to decision-making directly by voting. Long-lived services of this type would need to dynamically group devices or people based on their geographic locations, solicit input, carry out aggregations, and exhibit conditional behavior. We argue for the need to identify such concerns as purely communication/ interaction concerns, and to separate the mechanisms required to support them in separate layers or systems for better modularity and reusability.

There is a growing body of work separating interaction concerns from functional concerns (e.g., [3], [6]); however, the setting up of an interaction and making changes to it is cumbersome, orchestration of the interaction is still left to the interacting processes, and participants in an interaction are assumed to be known in advance.

Our approach is: to move the driving force for interactions between agents to the communication side of the divide, enabling interactions to drive themselves; to enable the setting up of rendezvous between agents at run-time, and treat it as a communication concern; to enable creation of libraries of novel types of communications with required aggregation and decision-making requirements, which can be launched and used by applications at run-time.

*This work is based on Hongxing Geng's Ph.D. research at the Agents Laboratory at University of Saskatchewan. He is also employed at Library and Scholarly Resources, Athabasca University.

Appears in: *Proc. of the 16th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2017), S. Das, E. Durfee, K. Larson, M. Winikoff (eds.), May 8–12, 2017, São Paulo, Brazil.*

2. INTERACTORS

Our model, interActors, is defined in terms of – and extends – the Actors model of concurrency [2]. Actors are primitive agents which encapsulate objects with threads of execution, and interact using asynchronous messages. interActors extend Actors with support for complex communications. A communication is made up of a set of outlets and handlers. The outlets are for the agents to connect to the communication, and handlers carry out the communication's logic. There are two types of outlets: *input* outlets for agents to send messages to communications, and *output* outlets for communication to send messages to agents.

A system based on interActors can be seen as having two layers: the Actor Layer and the Communication Layer (see Figure 1). Communications reside in the Communication Layer and the actors – serving as computational agents – reside in the Actor Layer. The black circle in the figure shows an output outlet, and the white circle an input outlet.

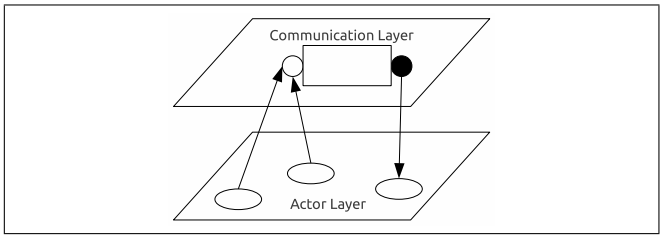

Figure 1: Two-layer Runtime System

Actors in the Actor Layer interact with a communication by sending messages to its input outlets and receiving messages from its output outlets. A communication's handlers are the driving force under the hood, and they can create more handlers or outlets, or change behaviors of outlets.

A primitive communication is called a *channel*, which simply connects an input outlet to an output outlet, to enable sending of asynchronous messages from an agent connect-

ing to the input outlet to another connecting to the output outlet. More complex communications can be created by composing channels using three rules: input merge, output merge, and output-input merge. An input merge composition combines a number of communications at their input outlets; the purpose is to enable a single sender to send messages to a number of recipients. An output merge merges a number of communications at their output outlets; the purpose is to enable a single output outlet to receive the messages from a number of sources. An output-input merge connects some output outlets of the composing communications to some input outlets.

This way of constructing complex communications is primarily for definitional purpose; complex communications can always be programmed directly.

Implementation. We have implemented a proof-of-concept prototype of interActors, which can be used to program new types of communications, and to compose them. Our implementation is in Scala [5] using the Akka actor library [7].

We have also developed CSL (Communication Specification Language), a specification language based on Scala, to ease programming of communications. CSL both dispenses with some of Scala's boilerplate, as well as restricts arbitrary computations from being included in a communication, with the intent of only allowing communications which can be constructed using the composition rules presented earlier.

3. CASE STUDY: US ELECTION

To illustrate the use of interActors, we show here how the interactions involved in an (admittedly simplified) US presidential election could be programmed as a communication.

At each polling station, a subset of eligible voters vote. Each vote is converted into a list, with 0s for all but one of the candidates, who receives 1. The polling station operates from a start time to a finish time, eventually sending the list of sums of all the votes to the county level.

At the county level, vote totals are received from each polling station in the county. These totals are aggregated as they arrive, and once all stations have reported, the aggregate is reported to the state level.

At the state level, similarly, county totals are received and aggregated; once all county totals have been received, the state's electoral college votes are awarded to the winner, and the results are sent out to the national level.

Finally, at the national level, electoral college votes received from the states are aggregated until the total number of electoral college votes for one of the candidates reaches 270, at which time the result is announced.

Programming It. It turns out that the types of individual communications we need for implementing this election are special cases of what we call a *multi-origin many-to-many communications* [4, 1]. This is when a number of parties want to send a collective message to some recipient(s), without any one taking the lead. Figure 2 shows how this can be programmed in CSL. First a number of attributes are specified: `participants` is the group of participants who want to send the collective message; `recipients` are the intended recipients of the message; `cond` and `aggr` are the condition and aggregation functions respectively, which determine when it is time to aggregate the messages collected thus far, and how the aggregation happens. Next, the outlets and handlers are specified. An output outlet `out` has behavior `forward` parametrized with `recipients` to forward the incoming messages to. Handler `aggrhandler` has behavior `aggregator` to which it passes the `cond` and `aggr` functions, as well as the name of the output outlet `out`. Input outlet `in` too has the `forwarder` behavior, but parametrized with the handler `aggrhandler`'s name to forward incoming messages from `participants` to `aggrhandler`.

```
1  communication MOM2M {
2    attributes: {
3      participants: List[ActorRef];
4      recipients: List[ActorRef];
5      cond: List[Any] => Boolean;
6      aggr: List[Any] => Any;
7    }
8    output outlet: out(forwarder(recipients));
9    handler: aggrhandler(aggregator(out, cond, aggr));
10   input outlet: in(forwarder(aggrhandler));
11   init: {
12     sendm(participants, in);
13   }
14 }
```

Figure 2: Multi-Origin Many-to-Many Communication

Constructing a complete election communication now simply requires composing several instances of `MOM2M` communications, each parameterized with appropriate `cond` and `aggr` functions to be received via setting of attributes.

An agent can then use such an election communication by instantiating it, setting its attributes, and finally launching it as shown in Figure 3.

```
1 val e = new election();
2
3 e.setAttr(Map(...,
4    "participants1"->parts1,
5    "cond_county1"->cond_c1,"cond_station1"->cond_s1,
6    "aggr_county1"->aggr_c1,"aggr_station1"->aggr_s1,
7     ...));
8
9 e.launch();
```

Figure 3: Instantiating and Launching Communication

4. CONCLUSION

In a variety of emerging multiagent applications, interactions are becoming more complex and varied, often requiring aggregation and decision-making at run-time. Leaving such complex interactions to be managed by the agents complicates the agents' code, and hampers reusability. Existing approaches to separating these interaction concerns create static protocols which cannot evolve over the course of an interaction. In this paper, we presented a model, interaActors, and related mechanisms, for first-class communications which are self-driven, and which can respond to the changing state of an interaction. We briefly described our implementation, and a specification language which programmers can use to specify executable communications. Finally, we presented a case study to illustrate the ease of with which communication can be programmed using interActors.

Acknowledgments

Support from NSERC is gratefully acknowledged.

REFERENCES

[1] A. Abdel Moamen and N. Jamali. Coordinating crowd-sourced services. In *Proceedings of the IEEE International Conference on Mobile Services*, pages 92–99, Alaska, USA, June 2014.

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

[3] F. Arbab. Reo: A Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(3):329–366, June 2004.

[4] 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)*, pages 81–86, New York, NY, USA, 2013. ACM.

[5] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language. Technical report, 2004.

[6] M. P. Singh. Information-driven Interaction-oriented Programming: BSPL, the Blindingly Simple Protocol Language. AAMAS '11, pages 491–498, Richland, SC, 2011.

[7] Typesafe. Akka Framework. `http://www.akka.io`.