CooS: Coordination Support for Mobile Collaborative Applications

Mario Henrique Cruz Torres, Robrecht Haesevoets, and Tom Holvoet

iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
{MarioHenrique.CruzTorres,Robrecht.Haesevoets,
Tom.Holvoet}@cs.kuleuven.be

Abstract. The advent of mobile devices, such as smartphones and tablets, and their integration with cloud computing is turning ubiquitous computing into reality. This ubiquity opens doors to innovative applications, where mobile devices collaborate on behalf of their users. Applications that leverage this new paradigm, however, have yet to reach the market. One of the reasons is due to the inherent complexity of developing such collaborative applications on mobile devices.

In this paper, we present a middleware that enables coordination on mobile devices. Our middleware frees applications from directly managing the interaction between collaboration partners. It also uses contextual information, such as location, to dynamically determine possible collaboration partners. We focus on a particular class of applications in which mobile devices have to collaborate to allocate tasks (e.g., picking up passengers) to physically distributed resources (e.g., taxis). The technical feasibility of our middleware is shown by the implementation of our middleware architecture, a deployment of our middleware on a real cloud environment and operating it with over 800 clients.

1 Introduction

Every day, developers create dozens of new applications for smartphones and other mobile devices. Two important trends are making mobile devices the platform of the future. First, they provide a hardware platform, filled with technology, that is getting cheaper everyday. Modern devices come with communication technologies, like Bluetooth, GPRS, EDGE, and WiFi, and an abundance of sensors, such as accelerometers, compasses, altimeters, and GPS (Global Positioning System). Second, with the advent of cloud computing, it is possible to create applications that scale to serve hundreds of thousands of clients, while providing minimum delays, needed for near real-time mobile collaborative applications. In fact, cloud computing is changing the way computing is offered. Computing power has become a utility that applications can consume at will, facilitating the deployment of large scale mobile collaborative applications. Ubiquitous computing [17] is finally reality due to the advent of mobile devices and cloud computing, opening doors to innovative applications.

One particularly promising type of applications, are applications where mobile devices closely collaborate, on behalf of their users. These applications include crowd sourcing internet connections [9], collaborative traffic routing [1], collaborative scheduling of resources (e.g., cars) [7, 11], search and rescue systems [8], or allocating

K. Zheng, M. Li, and H. Jiang (Eds.): MOBIQUITOUS 2012, LNICST 120, pp. 152–163, 2013.

taxis to passengers in a dial-a-ride problem [6]. In the resource sharing problem, users can use the location information from their mobile devices to collaborate with other users in their vicinity, to organize on-the-spot car pools, for instance. Another very useful application is for improving public transportation with the use of autonomous vehicles, that could collaborate to find the best way to pick passengers [11].

Despite today's pervasiveness of mobile devices and the challenging problems that could be addressed using collaborations, applications that truly leverage the power of collaboration on mobile devices are still missing. One of the main reasons for this lack of applications is due to the inherent complexity of developing such applications. Mobile collaboration may also require users coordination. Existing coordination mechanisms, such as ContractNet [15], or MASCOT [12], require specific interaction flows involving large amounts of messages between coordination partners. Ensuring the correct implementation and execution of such mechanisms can be time consuming and error prone. Another problem is that collaboration partners are often not known in advance, but have to be determined dynamically, for example, based on their location. In addition, all these problems take place in a very dynamic environment, where everybody is moving, and where disconnections and changes in commitment are widespread.

To stimulate the future development of mobile collaborative applications, we need good middleware support that relieves developers of such complexities. In this paper, we present CooS[1], a middleware that operates providing common-middleware services [14] that enable the creation of decentralized collaboration of mobile devices. CooS targets a particular class of applications in which mobile devices have to collaborate to allocate tasks (e.g., picking up passengers), to physically distributed resources (e.g., taxis, autonomous cars). CooS addresses three key challenges:

1. dynamically determining collaboration partners (e.g., based on their location),
2. achieving scalable collaborations,
3. managing the interactions between collaboration partners.

The main contribution of this paper is a middleware to enable the creation of large-scale mobile collaborative applications. The novelty of our approach is to integrate location-based participant selection with coordination mechanisms, and offering this functionality as a reusable middleware service. The middleware service is designed to be deployed on any cloud computing provider.

Overview. The remainder of this paper is organized as follows: Section 2 describes the challenges faced to create mobile collaborative applications. Section 3 details the design goals and Section 4 the architecture of our middleware. We describe experiments of an application developed on top of CooS, and analyze their results, at Section 5. Section 6 discusses related work. Finally, in Section 7 we present our conclusions.

2 Problem Statement

Our goal is to provide a middleware that supports the development of collaborative applications on mobile devices. Such applications typically require coordination of mobile devices to set up and execute the required collaborations. To illustrate the type

[1] CooS: Coordination on Clouds.

of applications we want to address, we focus on the dial-a-ride problem for taxis. In this problem mobile devices have to collaborate to allocate tasks (i.e., picking up and dropping off passengers), to physically distributed resources (i.e., taxis).

The dial-a-ride problem has been extensively studied due to its applicability in various domains. The problem is computationally demanding, even for small scale instances [10], and can involve various stakeholders with opposing goals. In the taxi problem, for example, taxi companies want to maximize their profit, typically at the expense of competing companies, and are even willing to compromise their quality of service (e.g., picking up a passenger on time). Passengers, however, want to be picked on time and reach their destination as soon as possible. The goal of the dial-a-ride problem is to pick up passengers in time, while maximizing the profit of all taxi companies.

Resources have a physical location and are mobile. Tasks are also location-based. Resources can commit to tasks (e.g., a taxi agreeing to pick up a passenger), de-commit to tasks (e.g., a taxi taking an alternate route), and can break down (e.g., a taxi breaking down). The number of involved resources and tasks can vary dynamically and scale up to thousands, for large collaborations.

In the rest of this section, we elaborate some key challenges in building mobile collaborative applications.

2.1 Key Challenges

Dynamically Determining Collaboration Partners Based on Location. Classic coordination mechanisms, such as ContractNet or auctions, do not take location into account when determining possible collaboration partners. In our taxi problem, this would result in mobile devices of passengers interacting with the devices of all taxis in the system to find a possible resource. This leads to our first challenge.

> **Challenge 1.** A device should only collaborate with those devices whose location fits within the solution space of the underlying problem.

In our taxi problem, the mobile device of a passenger should only collaborate with the devices of taxis that are within a feasible range to pick up the passenger. Since both taxis and passengers are mobile, collaboration partners can change dynamically.

Scalable Collaboration. Each mobile device, active in the system, will have a communication overhead. This overhead can be related to the actual collaborations a device is involved in, but also to the process of finding the right collaboration partners. While a device may only have to collaborate with a few dozen of other devices, there can be thousands of devices that are all potential collaboration partners. Finding the relevant collaboration partners may induce a communication overhead that is disproportionate to the overhead induced by the actual collaboration. This defines our second challenge.

> **Challenge 2.** The communication overhead of a device in the system, related to finding relevant collaboration partners, should be independent of the total number of devices in the system.

The communication overhead of each device in the system is only dependent on the number of devices it directly collaborates with.

Managing the Interactions between Collaboration Parters. Coordination mechanisms tend to get complex, requiring asynchronous interactions with complex message flows. Current technologies, such as GCMA (Google Cloud Messaging for Android) [2], only provide a basic messaging mechanism for the interaction of cloud services and mobile devices. Managing these interactions can be time consuming and error-prone. Reuse existing coordination mechanisms could greatly improve these problems. Achieving such reuse, however, requires a clean separation between application logic and coordination logic, which poses an even bigger problem. This leads to our final challenge.

> **Challenge 3.** Coordination mechanisms and their required interactions should be easy to manage, allowing developers to separate application logic from coordination logic, while promoting reuse of existing coordination mechanisms.

2.2 Requirements for the CooS Middleware

Given the challenges for developing mobile collaborative applications, we can derive a set of functional and non-functional requirements for the CooS middleware. There are two main functional requirements for the CooS middleware:

1. **Dynamic Partner Selection.** The middleware dynamically selects the relevant collaboration partners based on their location.
2. **Managing Interactions between Collaboration Partners.** The middleware enforces the coordination mechanisms, chosen by the application developer, ensuring the required interactions take place without violating message flows or timing constraints.

We can also derive two non-functional requirements for the CooS middleware:

1. **Scalable Partner Selection.** The middleware ensures that communication overhead, of each device, related to participant selection is independent of the number of devices in the system.
2. **Encapsulation of Coordination Mechanisms.** The middleware encapsulates the coordination mechanisms and related interactions as reusable middleware services. The middleware provides an API to application developers that allows to separate application logic from coordination logic.

3 Design of the CooS Middleware

Before explaining the CooS Middleware architecture in detail, we provide a high-level overview of its design and motivate the most important design decisions.

[2] `http://developer.android.com/guide/google/gcm/`

Providing Coordination Mechanisms as a Reusable Middleware Service. A key requirement of the CooS middleware is to manage the interactions between collaboration participants, relieving application developers from the related complexities. To do so, the middleware provides a set of predefined coordination mechanisms as reusable middleware services. Applications can then choose the proper coordination mechanisms according to their needs.

Using an Event-Driven Architecture to Enforce Coordination Mechanisms. To provide the coordination mechanisms, the middleware needs to enforce the required interactions between the collaboration partners. To do so, the CooS middleware relies on an event-driven architecture. Each coordination mechanism is defined as a set of interaction events (i.e., sending and receiving messages) that have to take place in a specific order and within particular timing constraints.

The event-driven architecture is particularly suited to handle the continuous internet connections and disconnections of mobile devices. It also allows to create a thin middleware layer to be deployed on mobile devices, which are typically computational constrained.

Using Location-Based Publish/Subscribe to Select Partners. Another key requirement of the CooS middleware is to dynamically select coordination partners based on their location. This avoids interaction with irrelevant participants, such as taxis in other cities. To achieve this dynamic partner selection, the CooS middleware employs a location-based Publish/Subscribe mechanism [2]. The location-based Pub/Sub system allows to subscribe to events, based on the location or region in which an event occurs. Every time a new event is created in a location, the subscribers to that location or region receive a notification. Publishers of events attach location information to their events, so this information can be used to match interested subscribers.

Using the location-based Pub/Sub system, the middleware notifies the relevant applications whenever a new collaboration is triggered within their regions of interest. To do so, the CooS middleware maintains the location of each mobile device active in the system.

Offloading Coordination-Specific Functionality to Mobile Devices. Providing coordination mechanisms and dynamically selecting coordination partners requires functionality such as determining the location of mobile devices, or calculating the shortest path from a passenger to a taxi. The CooS middleware relies on the capabilities of modern devices to offload these tasks to the devices themselves. The CooS middleware uses the GPS of the device, for example, to determine the location of taxis or passengers, and the locally available routing software to calculate possible paths.

Using a Cloud-Based Infrastructure. While mobile-devices can be used to offload some of the coordination-specific functionality, the actual enforcement of coordination mechanisms and selection of collaboration partners can put a heavy burden on the mobile devices, if done locally. To relieve the mobile devices, the CooS middleware relies on a cloud-based infrastructure to enforce the coordination mechanisms and to determine the possible collaboration participants.

The cloud-based infrastructure also provides a more uniform communication channel. Many times, mobile devices cannot communicate directly with each other, because they are situated behind proxies or firewalls. The cloud, however, is able to provide a uniform messaging layer to all mobile devices.

To provide additional scalability to applications, middleware services deployed on the cloud can easily be replicated to more computers. This allows to scale applications to match the current number of users.

4 CooS Architecture

The runtime architecture of the CooS middleware consists of two main components: the CooS Client Component, deployed on each mobile device, and the CooS Middleware Component, deployed on a cloud provider.

The CooS Middleware Component has two main responsibilities: (1) dynamically selecting relevant collaboration partners based on their location, and (2) enforcing a particular coordination mechanism, chosen by the application developer, among the selected partners.

The CooS Client Component serves as a mediator between the CooS Middleware Component and the application. It provides an API that allows the application use the coordination mechanisms provided by the CooS Middleware Component. A user has two possible ways of collaborating with other users, as an initiator, triggering a collaboration, or as a participant, waiting for collaboration requests, the CooS Client Component allows applications to play two possible roles: the initiator role or the participant role. In the taxi application, for example, the application plays the initiator role at the passenger's device, and the participant role at the taxi driver's device.

We illustrate the middleware architecture with ContractNet as coordination mechanism (Sect. 4.3), and briefly discuss the implementation of the CooS middleware architecture (Sect. 4.4).

4.1 CooS Middleware Component

The CooS Middleware Component uses an event-driven architecture to enforce its coordination mechanisms, and relies on a location-based Publish/Subscribe mechanism [2] to dynamically select the collaboration participants. The internal architecture of the CooS Middleware Component consists of four components: a Coordination Component a Location-Based Publish/Subscribe Component, a Location Store, and an Event Dispatcher (Fig. 1).

The Coordination Component is responsible for enforcing the selected coordination mechanisms among the active participants. This includes making sure that interaction events (i.e., sending or receiving message) take place in the right order without violating any timing constraints. The Coordination Component is also responsible for maintaining the state the ongoing coordinations. Coordination mechanisms can define constraints based on location information. The Coordination Component is a publisher and a subscriber of events from the Location-Based Publish/Subscribe Component.

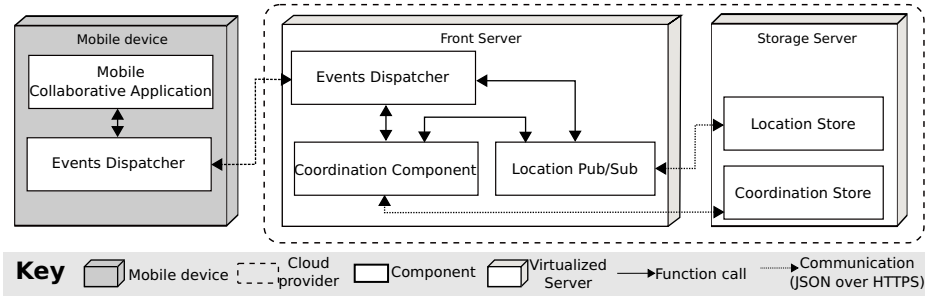

Fig. 1. Deployment view of CooS middleware on a cloud provider

The Location-Based Publish/Subscribe Component provides the functionality for location-based participant selection. Active devices publish their location information using the CooS Client Component. The location information is processed by the Location-Based Publish/Subscribe Component and persisted on secondary storage.

The Event Dispatcher is responsible for receiving events and dispatching events from and to the CooS Client Components. The Event Dispatcher relies on a unique *DeviceID* to identify each CooS Client Component, allowing to have asynchronous interactions between CooS Client Component and CooS Middleware Component. Interaction between the Event Dispatcher and the CooS Client Components is based on stateless protocols, such as HTTP.

4.2 CooS Client Component

The CooS Client Component acts as a mediator between the CooS Middleware Component and the application. It provides an asynchronous API to applications to use the coordination mechanisms provided by the CooS Middleware Component. The main API operations are illustrated below:

```
requestCollaboration(DeviceID device,
 Coordinates location,
 Payload payload,
 InitiatorCallback cb)

registerAsParticipant(LocationCallback lcb,
  ParticipantCallback pcb)
```

To start a collaboration the application uses the *requestCollaboration* operation of the CooS Client Component. The CooS Client Component, in turn, creates an event including the *DeviceID*, the location of the device, and an application-specific payload. The CooS Client Component then dispatches this event to the CooS Middleware Component. When invoking the *requestCollaboration*, the application needs to pass an *InitiatorCallback*. This callback is specific to the coordination mechanism, and provides the actual functionality of the application to be the initiator of the coordination. For example, when using the ContractNet coordination mechanism, the callback should provide the functionality to inform the application with the outcome of the ContractNet protocol. The *Payload* is application specific data not inspected by the middleware. The middleware only passes this data back to the application.

To participate in collaborations, applications have to register two callbacks, using the *registerAsParticipant* operation of the CooS Client Component. The first callback is the *LocationCallback*. This callback is responsible for providing the middleware with the proper location information, required by the location-based participant selection of the CooS Middleware Component. The second callback is the *ParticipantCallback*. Like the *InitiatorCallback*, this callback is specific to the coordination mechanism, and provides the actual functionality of the application to be a participant in the coordination.

4.3 Illustration of the CooS Middleware Architecture

To illustrate the CooS middleware architecture, we show how applications can register as participant and how applications can request collaborations. To register as participant, applications call the *registerAsParticipant* operation on the CooS Client Component (Fig. 2). The local CooS Client Component then starts a process that will retrieve the application-specific location on regular intervals from the application, and send location updates to the CooS Middleware Component. The CooS Middleware Component stores these locations in its location store. Once registered as a participant, the CooS middleware will take these applications into account when selecting the relevant collaboration partners for each new collaboration.

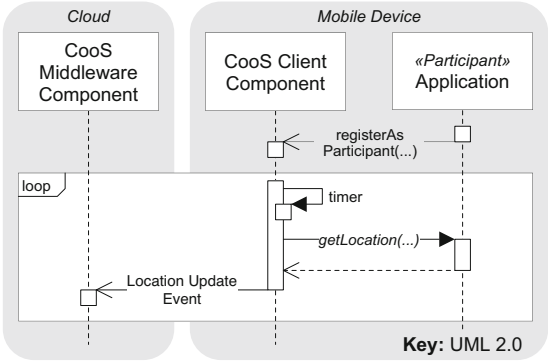

Fig. 2. A sequence diagram showing how the CooS middleware maintains the location of each potential collaboration participant

When an application starts a collaboration, it calls the *requestCollaboration* operation on the CooS Client Component (Fig. 3). The CooS Client Component sends this request to the CooS Middleware Component, which selects the relevant participants, among the registered applications, based on their stored location. The CooS Client Component of each selected participant is then informed about the collaboration request. These CooS Client Components will then start a coordination-specific interaction with their local application (in Fig. 3, this interaction is shown as generic *collaborationCalls* and *Coordination Events*). All interaction events between participants pass through the CooS Middleware Component, which uses the context of active coordination sessions to act as an interaction hub.

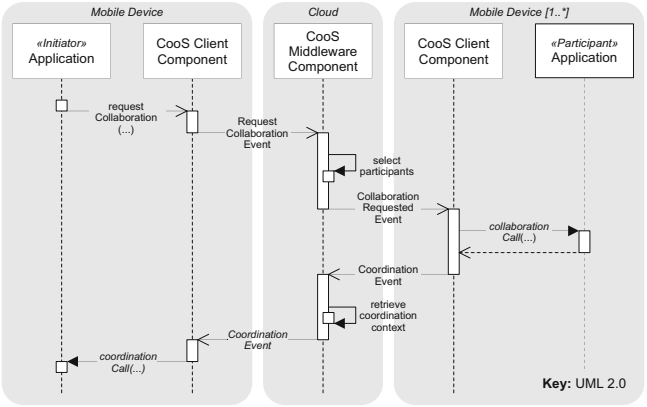

Fig. 3. A sequence diagram showing how an application can initiate a collaboration

4.4 Implementation

The CooS prototype uses off-the-shelf technologies. The CooS Middleware Component uses Node.js[3], a high-performance event-driven application server for networked applications. The CooS Middleware Component maintains location and on-going coordination information, which is stored on a mongoDB[4] database. MongoDB is a scalable, high-performance, open-source NoSQL database.

The CooS Client Component and CooS Middleware Component have bi-directional communication, so that the coordination interactions can happen, with the cloud notifying the mobile devices and vice-versa. The prototype communication is made using the WebSockets [3] protocol.

5 Evaluation

5.1 Case Study: Using Smartphones for Coordinating Taxis in Brussels

We performed a case study in order to evaluate the technical implications of using our middleware in a more realistic setting. We implemented a coordination application to coordinate all the taxis in Brussels on their task of picking passenger and delivering them at the requested locations.

Our goal with this case study was to check the technical feasibility of using our middleware for such problem. Coordinating taxis consists in allocating the taxi that can pick a passenger in the shortest time, that way minimizing the passenger waiting time. Passengers have the application installed on their mobile phones. When a passenger wants a ride, he simply indicates when he will need a taxi and where he wants to go. This information, together with the location information given by the GPS of the passenger's mobile device, is sent to all taxis that are interested in picking passenger and delivering them in a particular region.

[3] `http://nodejs.org/`
[4] `http://www.mongodb.org/`

5.2 Evaluation System Model

We have implemented a prototype version of our middleware, and deployed the **EventSignaling** part of our middleware on the Heroku[5] cloud provider.

We setup 80 computers to participate in the emulation, executing the taxi application. Every computer having 10 instances of the taxi application running as independent processes. Besides the taxi applications, we also setup 8 computers to simulate the passengers. Every computer executing 10 instances of the **PassengerApp**.

Hence, in our emulation we executed 880 instances of an application using our middleware. Each instance had a very simple simulator, responsible for issuing commands to the application. The commands consisted in simulating a taxi driver driving a taxi following a particular route and in passengers asking rides on their mobile phones.

We developed two simple components to simulate the behavior of a passenger and a taxi driver using our application. The simulators have the following behavior:

- **Passenger Simulator**, reads a location from the destinations list and asks a new ride to the *PassengerApp*. When the *PassengerApp* indicates the ride is done, the *Passenger Simulator* requests a new ride. Otherwise, if the *PassengerApp* indicates there is no taxi available, the *Passenger Simulator* chooses the following location from the destinations list and issues a new ride.
- **Driver Simulator**, simulates a taxi moving into the location of a passenger. It does this by virtually following a route given by the *TaxiApp*.

On a real world deployment of our application it would be possible to configure the location updates issued by the middleware to one update every few seconds, or more. However in our emulation we configured the middleware to issue a location update every 100 ms. What in our experiments lead to 8800 requests per second without any delay due to the number of requests. The application showed delays when handling more than 14.000 requests per second. The operation of the middleware at the client side is negligible, while the operations at the CooS Middleware Componentheavily relies on the performance of the cloud provider. The main shortcoming can be the response time due to the internet connection of the mobile devices.

Regarding the implementation of the taxi application, we learned that using the GPS (Global Positioning System) of mobile devices has to be done carefully in order to avoid draining the device's battery. Another lesson we learned from implementing the taxi client application is that delegating the communication complexity to the CooS middleware facilitated the application development, however it was still complex to manage all the callback functions needed by CooS.

6 Related Work

Our middleware does not deal with low level communication issues, instead it facilitates to coordinate the task allocation between several entities participating in an application. [14] proposes a layered view to position the different types of middleware

[5] `http://www.heroku.com`

available. Our work fits into the **Common Middleware Services** layer, since our middleware provides a higher-level domain-independent component that allows application developers to concentrate on programming application logic, rather than focusing on low level hurdles specific to the coordination protocol in use.

The work [5] adds quality-of-service guarantees to middleware which works upon the elastic resources from cloud computing. It shows a technique to guarantee a specified quality-of-service even on a changing cloud environment. In our evaluation we linearly increased the available cloud resources used by our middleware, in order to guarantee that all messages were properly delivered. Our middleware could integrate the results from [5] and become apt to work on more dynamic environments with sudden changes in usage.

There are several works exploiting middleware as a way mitigate different challenges associated with the application development for mobile devices [4], [13], [16], to cite a few.

The work [13] provides a number of abstractions to deal with mobile applications. The main goal of that work is to encapsulate the protocol behavior on well defined abstractions and to facilitate group formation of entities whom want to collaborate on a certain protocol. Our work does not deal with group formation, since we assume that service requests are sent to any device subscribed to the content of the service request. Our work focuses on facilitating the allocation of tasks between a number of mobile devices.

The development of mobile applications that leverage the cloud infrastructure is explored in [4]. [4] proposes a middleware capable of relocating specific parts of a application to be executed on the cloud, based on the quality criteria defined by the application developers. Our work also leverages from cloud computing, but we do not focus on optimizing the application execution. We focus on allowing the creation of collaborative applications on the mobile devices, leveraging the cloud as an infrastructure to interconnect the mobile devices.

7 Conclusion

In this paper, we presented CooS, a middleware that enables the creation of collaborative applications on mobile devices. CooS targets applications in which mobile devices have to collaborate to allocate tasks (e.g., picking-up passengers) to distributed physical resources (e.g., taxis). CooS addresses several key challenges for developing mobile collaborative applications. These challenges include dynamically determining collaboration partners, achieving scalable collaboration, and managing the interactions between collaboration partners.

We presented a middleware architecture for CooS that encapsulates coordination mechanism as a reusable middleware service for applications. This encapsulation provides a clear separation of concerns, freeing application developers from handling coordination-specific complexities. The evaluation of CooS showed the technical feasibility and scalability of the presented middleware architecture. As future work, we plan to perform an empirical study, with real software developers, to assess how CooS impacts the development of mobile collaborative applications.

Acknowledgments. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund KU Leuven.

References

1. Bazzan, A.: A distributed approach for coordination of traffic signal agents. Autonomous Agents and Multi-Agent Systems 10(1), 131–164 (2005)
2. Eugster, P.T., Garbinato, B., Holzer, A.: Location-based publish/subscribe. In: Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications, NCA 2005, pp. 279–282. IEEE Computer Society, Washington, DC (2005)
3. Fette, I., Melnikov, A.: The WebSocket Protocol. RFC 6455 (Proposed Standard) (December 2011), `http://www.ietf.org/rfc/rfc6455.txt`
4. Giurgiu, I., Riva, O., Juric, D., Krivulev, I., Alonso, G.: Calling the cloud: Enabling mobile phones as interfaces to cloud applications. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 83–102. Springer, Heidelberg (2009)
5. Hoffert, J., Schmidt, D.C., Gokhale, A.: Adapting distributed real-time and embedded pub-/sub middleware for cloud computing environments. In: Gupta, I., Mascolo, C. (eds.) Middleware 2010. LNCS, vol. 6452, pp. 21–41. Springer, Heidelberg (2010)
6. Koźlak, J., Créput, J.C., Hilaire, V., Koukam, A.: Multi-agent approach to dynamic pick-up and delivery problem with uncertain knowledge about future transport demands. Fundam. Inf. 71(1), 27–36 (2006)
7. Kutanoglu, E., Wu, S.: On combinatorial auction and lagrangean relaxation for distributed resource scheduling. IIE Transactions 31(9), 813–826 (1999)
8. Luqman, F., Griss, M.: Overseer: a mobile context-aware collaboration and task management system for disaster response. In: Eighth International Conference on Creating, Connecting and Collaborating through Computing, UC San Diego, La Jolla CA, United States (2010) (2010)
9. Papadopouli, M., Schulzrinne, H.: Connection sharing in an ad hoc wireless network among collaborating hosts. In: Proc. International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), pp. 169–185 (1999)
10. Parragh, S.N., Doerner, K.F., Hartl, R.F.: Variable neighborhood search for the dial-a-ride problem. Computers & Operations Research 37(6), 1129–1138 (2010)
11. Rocha, R., Cunha, A., Varandas, J., Dias, J.: Towards a new mobility concept for cities: architecture and programming of semi-autonomous electric vehicles. Industrial Robot: An International Journal 34(2), 142–149 (2007)
12. Sadeh, N., Hildum, D., Kjenstad, D., Tseng, A.: Mascot: an agent-based architecture for dynamic supply chain creation and coordination in the internet economy. Production Planning & Control 12(3), 212–223 (2001)
13. Schelfthout, K., Weyns, D., Holvoet, T.: Middleware for protocol-based coordination in mobile applications. IEEE Distributed Systems Online 7(8), 1–18 (2006)
14. Schmidt, D.C.: Middleware for real-timeand embedded systems. Communications of the ACM (2002)
15. Specification, F.: `http://www.fipa.org/specs/fipa00029.SC00029H.html` (2003)
16. Ueyama, J., Pinto, V.P.V., Madeira, E.R.M., Grace, P., Jonhson, T.M.M., Camargo, R.Y.: Exploiting a generic approach for constructing mobile device applications. In: COMSWARE 2009, pp. 12:1–12:12. ACM, New York (2009)
17. Weiser, M.: Some computer science issues in ubiquitous computing. Communications of the ACM 36(7), 75–84 (1993)