Application Design for Ad Hoc Collaboration Environment Based on Dependency Injection Pattern

Mirko Randić, Marijan Kunštić, Bruno Blašković
Faculty of Electrical Engineering and Computing,
University of Zagreb, Croatia
mirko.randic@fer.hr

Abstract—**In this paper we elaborate applicability of dependency injection pattern as a design solution for ad hoc network applications development. Dependency injection pattern promotes inversion of control. Inversion of control has to put the server to control the binding process. Our main idea was an architecture with a thin client and a server with more competences that promote competition of servers. We suppose that a server is interested to be occupied by servicing as much as possible. It discovers for possible engagements and optionally makes a dependency injection to a client. This is about servers that compete for an engagement, not about clients that find servers. Static and dynamical aspects of the application are specified by the UML. For the system implementation we used the LIME middleware and Java technology.**

I. Introduction

Ad-hoc network is a world of hosts capable of wireless communication, moving freely through space, computing in a highly decoupled style, and interacting opportunistically. In [1] is stated: the term *ad hoc network* describes a spontaneous communications structure with a dynamic topology and self-organizing capability. In [2], ad hoc network is described as dynamic environment that exhibits transient interactions, decoupled computing, physical mobility of hosts, and logical mobility of code. Any way, ad hoc networks represent a complex technology that poses many challenges to applications development. New coordination models, dynamic binding strategies at the application level and design solution need to be considered and suitable techniques developed. These were main motivations for the work presented in this paper.

The following section presents a background relevant for this work. In Section III we present overall view on static and dynamical aspects of an application that incorporates inversion of control. Dependency injection pattern and application's static structure are described and presented by UML class diagrams. Collaboration logic based on service announcement and engagement discovery processes are described too. Section IV specifies dynamics of a client and server in more details (state transitions and interactions). Section V contains conclusions.

II. Background

In [3] a flexible binding approach that comprehends service discovering, following and binding on the fly is elaborated. A new concept is introduced that considers the issue of automatic and transparent discovery of services and the maintenance of channels between an application and the services needed to carry it out. The concept is called *context-sensitive binding*; a way to maintain the service provision channel between two entities dynamically as they move through changing physical and logical contexts. Context-sensitive binding efficiently decouples the interface of the service from its physical realization. In [4] the context-sensitive binding concept is defined. "*While context-sensitive binding shares some commonality with other approaches to dynamic binding (the idea of making binding decision at run-time), the novelty it brings lies in the fact that it does not make this binding decision once but on a continuous basis in direct response to changes in the operational environment.*" Client is responsible for coordination the context-sensitive binding process. Our approach puts a server into the center of binding control. This inversion of control that promotes dependency injection implies some new characteristics of collaboration between components in ad hoc system. Injection represents a kind of configuration mechanism for direct configuration.

One of the key features of service oriented computing, SOC [5] is the decoupling between the interface and the implementation of some functionality, which can be advertised by service providers and discovered by service users at run-time. Service oriented computing comprises three salient elements: *service specification*, *service request announcement* and *service advertisement*. Service specification is responsible for describing a service in a comprehensive and unambiguous manner that is machine interpretable to facilitate automation. Service request announcement is responsible for announcing a given service specification by a service user (a client) on a media. Client formulates a request (a specification of its needs) that is formatted in a similar manner to the service specification. Service advertisement is responsible for advertising a given service specification by a service provider on a media. Advertisement has to provide information about offered service that can help a user to determine whether she (it) would like to exploit that service. SOC promotes service discovery process controlled by user, while our approach promotes engagement discovery process controlled by service provider.

One common coordination mechanism in distributed systems is shared memory, or more precisely shared data structures. Generally, shared data spaces ensure a great degree of *components decoupling both in time and space* that is of paramount importance for distributed applications in ad hoc systems. LIME [6] is a Java implementation of the Linda [7] coordination model, adapted for ad hoc networking. LIME represents sophisticated transiently shared tuple (data) space where individual tuple spaces can be shared or separated as changes in connectivity between hosts occur. We use the LIME technology in our system design and implementation.

III. Overall view on system design

A. Dependency injection pattern

In the world of object-oriented applications, dependency injection represents a pattern which help to decouple the client object from service implementation objects [8], [9]. It transfers the binding control to the server side too. This means that newly constructed service implementation object is offered or even imposed to the client. The pattern envisages existence of one assembler object responsible for creating a service implementation object following precise specification and finally for making an assembly consists of the client and server (Fig. 1). The assembler injects a reference to previously composed server into the client - *dependency injection.*

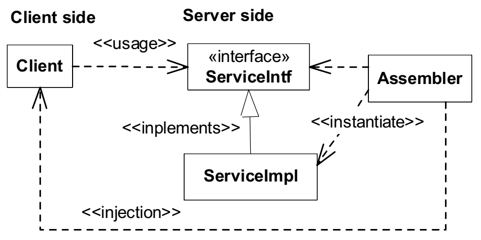

Figure 1. Dependency injection pattern.

Considering ad hoc computing environment, the injection pattern mechanism can be used to provoke competition of the servers. A client present in some area only has to make an announcement for the required service supplemented with some contextual (additional) information like language preferences, presentation format, etc. It is clear that some announcement media is needed. Servers in the area or just entering the area can "read" the announcements and if they are capable to offer requested service they dynamically compose adequate object collaboration. The collaboration is tailored to meet client additional request about preferences, abilities etc.

B. Application structure

Structure presented in the Fig. 2 provides necessary decoupling i.e. client code is independent in space and time of the concrete server implementation. Furthermore, client and server are both reactive. They react to the events of putting tuples into the shared tuple space. Their state transition dynamics is explained in Section IV. All event listener classes implement the `lime.ReactionListener` interface and provide definitions for `reactsTo(ReactionEvent)` operation. A server reacts when announcement or invocation request tuple is put into the shared space by a client. Generally, information contained in the announcement should enable: estimation of the server's ability to offer the service and server self-configuration. The announcement tuple has the form:

```
<clientID, bindingPolicy, serviceType,
    serviceFeature, velocityVector>.
```

Other valuable context information can also be a part of the announcement. Reaction to the announcement results in a complex server behavior described in subsection C. Client reacts to server's injection, advertisement, result and departing tuples.

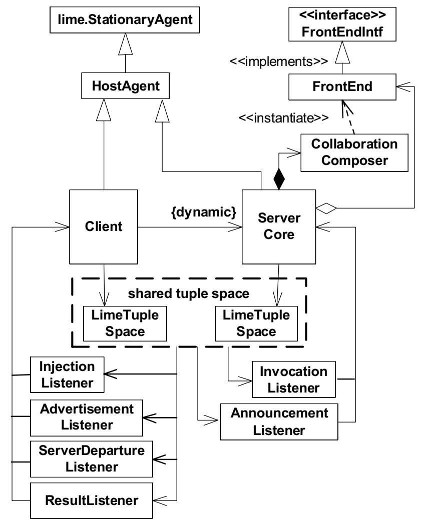

Figure 2. Application structure.

Responsibilities of the classes depicted in Fig. 2 are as follows:

- `Client` - makes an announcement or service invocation via shared tuple space. Eventually it can make service invocation directly via dynamically bound link to server object. Client reacts to events as mentioned before.
- `ServerCore` - tests own ability to offer required service. Initiates customized collaboration composition process relying on `CollaborationComposer` object. Makes dependency injection or advertises itself. When engaged, returns results to the client indirectly via shared tuple space or directly via dynamic link. Reacts to events as mentioned before.
- `CollaborationComposer` - dynamically creates a collaboration customized in accordance with client preferences. This collaboration really serves a client.
- For responsibilities of the `lime.StationaryAgent` and `lime.SharedTupleSpace` classes see [6].

In our system design, dependency injection is realized by shared tuple space and reaction mechanism. We call this *dependency injection via tuple space.* We have tested two types of information on which the dependency can be based: plain string Id and application level handle. String Id enables binding and further indirect service invocation via shared tuple space. On the other hand, handle should enable direct asynchronous service method invocations. Usage of the application level handle in ad hoc networks imposes usage of additional middleware which supports some kind of uni, multi or omnihandles and suitable communication framework. We experimentally use the Anhinga middleware [10]. In UML class diagram (Fig. 2) this kind of dynamic binding is rendered as an uni-directional association with constraint {dynamic} on the ServerCore association end. Semantics of the constraint is specified in [11].

C. Collaboration at the application level

Service oriented computing is based on service discovery (Fig. 3a). Service discovery is the process by which a user (client) finds an adequate service provider. Service discovery is essentially a two-part process of: advertising the provider and requesting by the user. Without the user's request, there is no (relation) binding established between the user and provider.

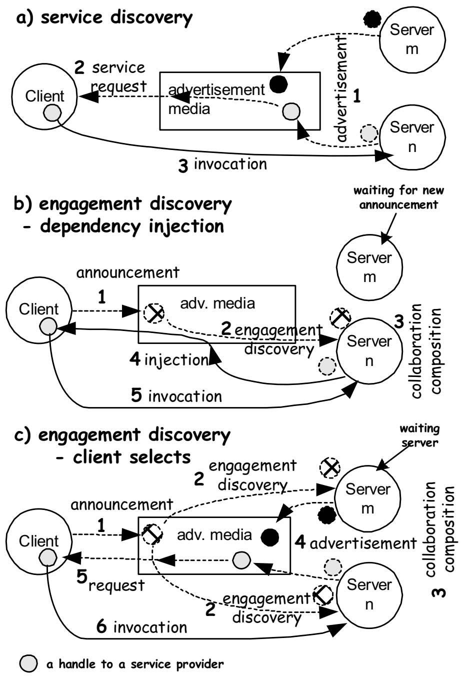

Figure 3. Service vs. engagement discovery process.

Our presumption was that a server is interested to be occupied by servicing. More occupations mean greater income. Therefore, in our collaboration model a server has an initiative; it tries to find an engagement and to inject dependency into a client.

In our approach binding is based on two processes: service announcement and engagement discovery. The client makes an announcement for the service it is looking for. Announcement means putting a public notice into a tuple space. Reaction to the client's announcement results in starting the engagement discovery process. Engagement discovery is the process by which a server is finding a potential client. First, server tests own ability to offer requested service. Server starts examination of the serviceFeature and velocityVector information from the announcement tuple. If it can offer requested feature for reasonable period of time it transfers the serviceFeature information to collaboration composer object. Collaboration composer dynamically creates customized collaboration and aggregates the collaboration into the ServerCore object. Finally depending on bindingPolicy specified by client the server injects its own Id or handle into the client via injection tuple (Fig. 3b) or puts an advertisement tuple into the shared space (Fig. 3c). The first option allows that fastest processing server or the server which first enters the client area (generally, the most agile server) to become engaged server. Surely, such collaboration logic implies severe security problems that have to be resolved. The second option includes server advertisement. A client reacts to all relevant advertisements and decides on the best one. Generally, one server in the area will get a job and start servicing, but others will reach the WaitingForEngagement state and wait for the opportunity for start servicing (e.g. if momentary engaged server goes off the area).

IV. The Client and Server Dynamics Modeling

A. State transitions

Statechart diagram is especially useful in modeling reactive or event driven entities just like the client and server objects.

Two states characterize a client: NotBinded and Binded (Fig. 4). Initially, client enters NotBinded state. "Putting a tuple in space" kinds of events that can initiate state transition are: injection, advertisement and departing that are handled with corresponding reaction listeners. On transition client can send a signal to server by putting adequate tuple into the shared space. Every time when client enters NotBinded state an announcement is sent. NotBinded state has an internal transition (see [12]) with advertisement event as a trigger and guard [bindingPolicy = noInjection && advertisement = nonAcceptable]. This event will be handled without living the state and an entry action will not be dispatched. When client is already binded by dependency injection, any further injection attempts must be ignored.

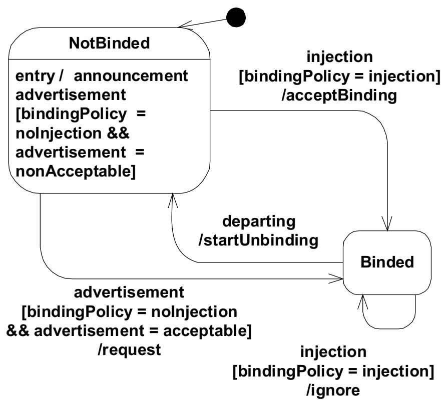
NotBinded
entry / announcement
advertisement [bindingPolicy = noInjection && advertisement = nonAcceptable]
injection [bindingPolicy = injection] /acceptBinding
departing /startUnbinding
Binded
advertisement [bindingPolicy = noInjection && advertisement = acceptable] /request
injection [bindingPolicy = injection] /ignore

Figure 4. Client's statechart diagram.

Fig. 5 depicts that a server can receive only two different signals from a client. As we said before, signal here represents "putting a tuple" event. Depending on transition, server can send advertisement, injection or departing signals to client. The injection signal is sent to particular client.

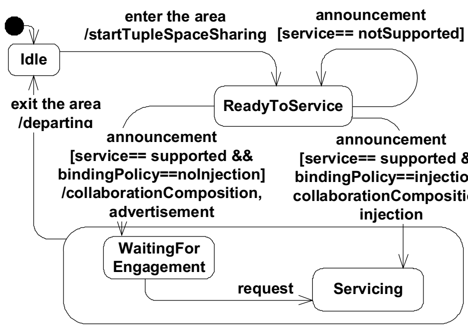

Figure 5. Server's statechart diagram.

B. *Interactions*

Fig. 6 depicts interaction in the case where client's `bindingPolicy` is set to `noInjection` and handle type binding is established. In any case, applications for ad hoc networks are characterized by transient interactions. Here we don't speak about ordinary transient interaction where transient behavior is under control of the application. There are adequate constructs in UML for rendering such a behavior. Here *transient interaction* means an interaction with transient object where transient behavior is under control of external forces i.e. is a part of a context. Standard UML does not offer a suitable notation that will capture the notion of physical mobility and transient interactions among mobile hosts. In Fig. 6 we used a nonstandard kind of object lifelines to capture the flavor of transient interactions.

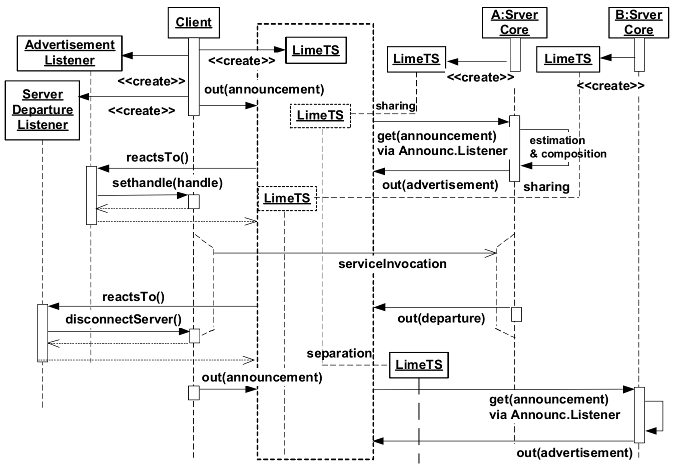

Figure 6. Interaction between a client and server objects.

The client in one moment puts an announcement in tuple space by calling `out()` method on `LimeTupleSpace` object. In this interaction only two client's listener objects are included: advertisement and server departure which are responsible for processing the events of putting advertisement and departure tuples. In the moment when any client and server meet in communication range the process of its local tuple space sharing is started. The mechanism for the process is implemented in the LIME middleware. We simply use this mechanism. After sharing spaces, server's `AnnouncementListener` object (not rendered in Fig. 6) automatically handles the announcement and delivers the announcement tuple to the server. Server analyzes announcements, estimates own ability and period of time it will be capable to service the client and start the customized collaboration composition process, prepare an advertisement and put out the advertisement in the shared tuple space by calling `out(advertisement)` method. Putting the advertisement tuple triggers event and `reactsTo()` method on `AdvertisementListener` object is automatically called. Advertisement tuple contains the server handle and client, if decide to employ this server, can immediately start usage of its service. Otherwise, client can wait for another advertisement, and previous server is put in `WaitingForEngagement` state. Interaction diagram depicts two server objects competing for an engagement. Object `A` arrives first and will leave the area before object `B`, so both object will have chance to serve the client.

V. Conclusion

Binding things together in ad hoc environment is difficult task. Application design for ad hoc collaboration environment is a special challenge. Work described in this paper is inspired by inversion of control principle that puts a server in active and competitive position. We have developed a prototypical application to test our design approach and found that dependency injection pattern combined with shared tuple space represent an efficient design option. Our previous work on UML based specification [11] is continued in this paper too. We are trying to find a suitable notation that will capture the notion for the context inhibited transient interactions.

References

[1] J. Wu, and I. Stojmenovic, "Ad hoc networks", *IEEE Computer* 18 (2004) pp. 29–31.

[2] R. Sen, R. Hondorean, G. C. Roman, and C. Gill, "Service Oriented Computing Imperatives in Ad Hoc Wireless Settings", Technical Report WU-CSE-2004-05, Washington University, Department of Computer Science, St. Louis, Missouri, 2004.

[3] R. Hondorean, R. Sen, G. Hackmann, and G. C. Roman, "Context Aware Session Management for Services in Ad Hoc Networks", Tech. Report WUCSE-2004-34,Washington University, Dept. of Comp. Science, 2004.

[4] R. Sen, and G. C. Roman, "Context-Sensitive Binding: Flexible Programming Using Transparent Context Maintenance", Tech. Report WUCSE-03-72, Washington University, Dept. of Comp. Science, 2003.

[5] Singh Munindar P. and Michael N. Huhns, *Service-Oriented Computing, Semantics, Processes, Agents, John Willey and Sons, 2005.*

[6] A. Murphy, G. Pico, and G. C. Roman, "LIME: A middleware for physical and logical mobility", Proceedings of the 21st International Conference on Distributed Computing Systems, pp. 524-533, 2001.

[7] D. Gerlenter, "Generative communication in Linda", ACM Computing Surveys, 7, 80-112, January 1985.

[8] M. Fowler, "Inversion of Control Containers and the Dependency Injection Pattern", 2004. http://www.martinfowler.com/.

[9] M. Fowler, "What is the difference between dependency and association?", 2003. http://www.martinfowler.com/.

[10] The Anhinga Project, Rochester Institute of Technology, http://www.cs.rit.edu/~anhinga/.

[11] M. Randić, B. Blašković, and P. Knežević, "Modeling Service Dependencies in Ad Hoc Collaborative Systems", Proceedings of the EUROCON 2005., pp.1842-1845, Belgrade, November 2005.

[12] OMG Unified Modeling Language Specification v 1.5, formal/03-03-03, March 2003.