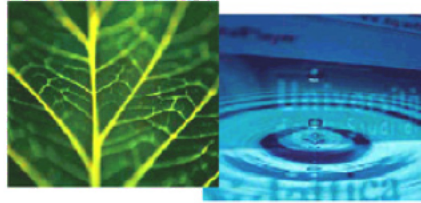


PhD Dissertation



**International Doctorate School in Information and
Communication Technologies**

DISI - University of Trento

**TOWARDS THE APPLICATION OF
INTERACTION-ORIENTED FRAMEWORKS TO
INFORMATION SHARING IN
EMERGENCY CONTEXTS**

Gaia Trearichi

Advisor:

Prof. Maurizio Marchese

Università degli Studi di Trento

December 2010

Abstract

In distributed, open environments, possibly heterogeneous computational entities need to engage in complex interactions in order to complete tasks and often have to face sudden changes; it therefore becomes essential for modern information systems to adopt coordination technologies which support dynamic and flexible interactions among processes, whether reactive (e.g., web services) or proactive (e.g., autonomous agents). Substantial efforts are being put forward to devise suitable mechanisms for process coordination. In the past few years, interaction-oriented frameworks have been proposed, which enable distributed and heterogeneous agents to engage in coordination activities by sharing interaction models specified in executable protocol languages. Software systems have started to be developed to apply such frameworks to concrete use. In particular, the OpenKnowledge framework has been proposed as such an interaction-oriented framework, and the OpenKnowledge (OK) system has been developed for its realization. Such system provides a distributed infrastructure which allows a-priori unknown peers to gather together and coordinate with each other by publishing, discovering and executing interaction models specified in the Lightweight Coordination Calculus (LCC) protocol language. Although the realization of the OpenKnowledge approach is promising, its application in realistic, complex scenarios is not fully exploited.

This thesis aims at applying the OpenKnowledge framework to realistic contexts such as emergency response (e-Response). Its main contribution is in the design and simulation of emergency response scenarios which are expressed in terms of LCC specifications, and are enacted by means of a simulation environment fully integrated with the OK system. Such envi-

ronment is developed to: (1) informally validate the e-Response scenarios; (2) test the capability of the OK system to support such scenarios, and (3) provide a preliminary evaluation of the efficacy of different information gathering strategies (i.e., centralized or distributed) in emergency response settings. The results obtained show that the OK system is able to support complex coordination tasks; however, some limitations have emerged in relation to the discovery mechanism. Furthermore, simulations have shown to adhere with realistic scenarios, and that - under ideal conditions - centralized and decentralized information-gathering strategies are comparable.

Keywords

[interaction models, peer-to-peer information sharing, agent-based disaster simulation]

Publications

This work has been developed in collaboration with various people (as the publications indicate) and in particular with: Maurizio Marchese, Lorenzino Vaccari, Veronica Rizzi, Juan Pane, Paolo Besana, Fiona McNeill, Nardine Osman.

Part of the material of the thesis has been published as articles in various conferences and journals and as technical reports in the EU FP6 OpenKnowledge¹ Specific European Targeted Research Project (STREP) project IST-FP11V341. In what follows, journal articles, conference papers, and technical reports are listed.

Journal articles (in order of appearance):

- [111]: Gaia Trecarichi, Veronica Rizzi, Maurizio Marchese, Lorenzino Vaccari, Paolo Besana. Enabling information gathering patterns for emergency response with the OpenKnowledge system, *Special Issue on Pervasive Computing and Application, Computing and Informatics Journal*, Vol. 29, N. 4, 2010, pp. 537-554.
- [70]: Maurizio Marchese, Lorenzino Vaccari, Gaia Trecarichi, Nardine Osman, Fiona McNeill, and Paolo Besana. An Interaction-Centric Approach to Support Peer Coordination in Distributed Emergency Response Management, *Intelligent Decision Technologies (IDT), Special Issue on Incident Management*, Vol. 3, N. 1, 2009, pp. 19-34.

Conference papers (in order of appearance):

- [112]: Gaia Trecarichi, Veronica Rizzi, Lorenzino Vaccari, Maurizio Marchese, and Paolo Besana. OpenKnowledge at work: exploring

¹<http://www.openk.org/>

centralized and decentralized information gathering in emergency contexts. *In Proceeding of the 6th International Conference on Information Systems for Crisis Response and Management (ISCRAM)*, 2009.

- [69]: Maurizio Marchese, Lorenzino Vaccari, Gaia Trecarichi, Nardine Osman, and Fiona McNeill. Interaction models to support peer coordination in crisis management. *In Proceedings of the 5th International Conference on Information Systems for Crisis Response and Management (ISCRAM)*, pages 230-241, 2008.

Technical reports (in order of appearance):

- [113]: Gaia Trecarichi, Veronica Rizzi, Lorenzino Vaccari, Juan Pane, and Maurizio Marchese. OpenKnowledge deliverable 6.8: Summative report on use of OpenKnowledge approach in eResponse: integration and evaluation results. *Technical report, OpenKnowledge*, 2008.
- [21]: Paolo Besana, Fiona McNeill, Fausto Giunchiglia, Lorenzino Vaccari, Gaia Trecarichi, and Juan Pane. Web service integration via matching of interaction specifications. *Technical report, University of Trento, Dipartimento di Ingegneria e Scienza dell'Informazione (DISI)*, 2008.
- [71]: Maurizio Marchese, Lorenzino Vaccari, Gaia Trecarichi, Pavel Shvaiko, Juan Pane, Nardine Osman, and Fiona McNeill. OpenKnowledge deliverable 6.7: Interaction models for eResponse. *Technical report, OpenKnowledge*, 2008.

Whenever results of any of these works are reported, proper citations are made in the body of the thesis.

Contents

1	Introduction	1
1.1	The problem	3
1.2	The proposed approach	5
1.3	Thesis structure	8
I	State of the Art	11
2	Service Coordination	13
2.1	Composition techniques	15
2.1.1	Workflow-based composition	15
2.1.2	Dynamic composition	16
2.2	Collaboration techniques	17
2.2.1	WSCI	18
2.2.2	WS-CDL	18
2.2.3	Let's Dance	19
2.2.4	BPEL4Chor	20
3	Agent-Oriented Coordination	21
3.1	Agent communication languages	22
3.2	Conversation policies	24
3.3	Social agency	26
3.3.1	Commitment protocols	26

3.3.2	Normative systems	27
3.3.3	Dialogue-game protocols	27
3.3.4	Electronic Institutions	28
3.3.5	Lightweight Coordination Calculus (LCC)	29

II An Interaction-Oriented Approach to Knowledge Sharing: OpenKnowledge 37

4 The Framework 39

5 The OpenKnowledge System 43

5.1	LCC interaction models	43
5.2	The OpenKnowledge Kernel	45
5.2.1	OKC components	46
5.2.2	Kernel modules	49
5.2.3	Kernel services	51
5.3	Interaction lifecycle	53
5.3.1	Subscription to an interaction model	53
5.3.2	Interaction bootstrap	54
5.3.3	Interaction run	55
5.4	OpenKnowledge vs. other technologies	56
5.4.1	Web services	57
5.4.2	Grid services	58
5.4.3	P2P systems	59
5.4.4	Multi-Agent Systems (MAS)	59

III Design and Simulation of LCC e-Response Scenarios 61

6 Information Sharing in Crisis Response 63

6.1	The general context	64
6.2	Flooding in Trentino: the e-response case study	67
6.2.1	Prealarm scenario	69
6.2.2	Evacuation scenario	71
6.3	Conclusion	74
7	LCC Protocol Design	77
7.1	Initial design	78
7.2	Final design	82
7.3	Conclusion	85
8	The Simulation Environment	87
8.1	LCC protocols: an overview	88
8.2	Simulation architecture	89
8.2.1	The peer network	90
8.2.2	The e-Response simulator	97
8.3	OK component reuse	108
8.3.1	Reuse of LCC interaction models	109
8.3.2	Reuse of OKC methods	110
8.4	Conclusion	111
IV	Experimental Evaluation	113
9	Experimental Testbed	115
9.1	Testbed overview	115
9.2	Experimental hypotheses	117
9.3	Experimental design	118
9.3.1	Performance measures	118
9.3.2	Experimental variables	118

9.3.3	Assumptions	121
9.3.4	Experiment configuration	121
9.3.5	Experiment execution	122
9.4	Results	122
9.4.1	Information gathering strategies	123
9.4.2	The OK infrastructure	125
9.5	Conclusion	128
V	Final Discussions	131
10	Related Work	133
11	Conclusions	137
12	Future Work	141
	Bibliography	145
A	Peer Network Interaction Models	165
A.1	Evacuation.lcc	165
A.2	Find-Route.lcc	169
A.3	Check-Route-State.lcc	170
A.4	Querier-Reporter.lcc	173
B	Simulator Interaction Models	177
B.1	Simulation_Cycles.lcc	177
B.2	Flood Sub-Simulator_Connection.lcc	182
B.3	Peer_Connection.lcc	183
B.4	Flood.lcc	187
B.5	Sensory_Info.lcc	189
B.6	Visualiser.lcc	191

B.7 Perform_Action.lcc	193
----------------------------------	-----

List of Tables

- 2.1 Collaboration vs. Composition 14
- 5.1 OpenKnowledge vs. Other Technologies 60
- 9.1 Experiments configuration 122

List of Figures

3.1	LCC formal syntax	30
3.2	LCC example: double arrows (\Rightarrow , \Leftarrow) indicate message passing, single arrow (\leftarrow) indicates constraint satisfaction.	31
5.1	Simple LCC interaction.	45
5.2	OK Kernel Architecture [34]	46
5.3	Annotated peer's methods	48
5.4	Peer's methods required by an OKC	49
5.5	OKC: access to peer's local knowledge	49
5.6	Peer Modules [34]	50
5.7	Interaction Subscription	54
5.8	OK network at subscription time [111]	55
5.9	Interaction Bootstrap	56
5.10	Interaction Run	57
6.1	Integrated Emergency Response Framework (iERF) proposed by NIST	66
6.2	The overall e-Response use case	68
6.3	The implemented e-Response scenarios	69
6.4	Evacuation phase: moving peer behaviour	75
7.1	UML activity diagram: baseline scenario	78
7.2	UML sequence diagram: baseline scenario	79
7.3	LCC fragment: <i>ES</i> initial role	80

7.4	LCC fragment: <i>ES</i> “route finder” role	80
7.5	UML activity diagram: evacuation scenario	83
7.6	LCC fragment for the “goal-achiever” role	84
7.7	LCC fragment for the “free-path-finder” role	84
8.1	e-Response Interaction Models	89
8.2	The e-Response system’s architecture	90
8.3	Evacuation phase: network peer’s interactions	91
8.4	LCC fragment for the “free-path-finder” role	93
8.5	Java code for OKC method “ <i>request_path_state</i> ”: interaction model enaction	94
8.6	Information Gathering: centralized interactions	96
8.7	Decentralized Information Gathering: selection of reporters	98
8.8	LCC fragment for the “info-handler” role taken by the con- troller	100
8.9	Maximum Water Level [2]	101
8.10	Maximum Flow Velocity [2]	101
8.11	Maximum impulse [2]	102
8.12	Maximum water level rising speed [2]	102
8.13	Arrival time of first water [2]	102
8.14	Maximum water level in Trento Nord	103
8.15	Arrival time of maximum water level	103
8.16	Flooding Law	104
8.17	Emergency GUI	106
8.18	Flood Evolution	107
8.19	GUI: Centralized information gathering	108
8.20	GUI: Decentralized information gathering	108
8.21	The “Querier-Reporter” IM Reuse	109
9.1	Percentage of arrivals by IG strategies	124

9.2 Outcome Distribution (centralized/decentralized scenario) 124
9.3 Time-steps vs. Path Length 125
9.4 Successful vs. Failed Experiment Runs 126
9.5 Failed experiment runs 127
9.6 Subscription failures by simulation run phase 128
9.7 Subscription failures by IMs 129

Chapter 1

Introduction

The need for effective process coordination has long been recognized in diverse research fields, such as service-oriented computing and agent systems. In open and distributed environments, possibly heterogeneous processes (whether web services or autonomous agents) must interact with each other in a way that allows the execution of complex coordination tasks. Two different realistic scenarios are briefly outlined below, to show the kind of issues involved in such settings.

In a possible e-tourism scenario, a customer wants to book a journey and forwards a request to a travel company able to deliver e-tickets. The company's information system supports process automation and integration with airline, car rental and hotel partners. When a request is received, the process coordinates different activities involved in the reservation procedure; for instance, it invokes services provided by the partner companies, stores information properly and handles new and unexpected situations (e.g., a partner service changes its interface; the customer changes its request, cancels reservations, asks for refunds; the company itself adds new services).

In a possible e-response scenario, a large number of actors (e.g., local governments, emergency coordination centres, fire brigades, the police and

health agencies, volunteers, citizens) are involved in the emergency activities. For instance, people in a coordination center may need to retrieve and integrate data and services from different geographical information systems in order to plan activities, make decisions and forward directives to their subordinates; geographically dispersed agents have to collaborate and coordinate in the disaster scenes by exchanging and reporting information with each other and with the people in the control room; moreover, the information distribution system has to cope with unexpected situations (e.g., actors changing roles, taking new and interrupting old activities).

In the e-tourism scenario, the Web is the scene where all the involved activities will take place. Here, the role of services is essential: they must be coordinated in order to provide more complex services and to respond to business requirement changes (e.g., partner and customer changes). The e-response scenario is much more complex and dynamic: beside the crucial problem of coordinating services (e.g., the problem of geo-service chaining), coordination mechanisms must support mobile agents located in areas prone to sudden changes. However, in both situations, the following key issues need to be addressed:

- Design of large-scale information systems (e.g., both scenarios involve many parties which are geographically dispersed);
- Dynamic and flexible interaction patterns (e.g., a customer may entertain long-lived interactions, emergency agents may engage in complex interactions or suddenly change the progress of their coordinated activities);
- Adaptive coordination mechanisms (e.g., one partner may change its protocol interface, an agency may deal with different organizations having different policies and procedures).

The above issues are recognized as top priorities among research communities. For example, Singh et al. [106] highlight the importance of adopting suitable software paradigms for “programming in the large”, i.e., the need for approaches able to model systems where large software components - built by different organizations over a long period of time, and having their local state - hold interactions which are subject to unforeseen events. Kang et al. [63] distinguish orchestration from choreography approaches: the former oriented in describing how composite processes are implemented and the latter focused on the interactions enacted by the processes without any central entity. Buhler et al. [25] make a distinction between the internal behaviour of the software components and their interactions: *“a software system is viewed as an ensemble of coordinables and their orchestrated interactions. Coordinables are entities that function as independent units of computation. The coordinated interaction of the computational units produces the desired behavior of the system”*.

All the above mentioned works put in evidence the role played by coordination mechanisms in software system design. Coordination technologies are, therefore, fundamental for supporting effective interactions between heterogeneous and distributed computational entities.

1.1 The problem

The issue of devising suitable coordination technologies is being tackled by several research communities: works on service-oriented computing and agent systems are converging to enable dynamic interaction among autonomous components in large, open systems.

On the one hand, the paradigm of Service-Oriented Architecture (SOA) and, as a consequence, that of Web services, is rapidly being accepted as the standard for coordinating distributed processes across networks, like

Internet.

On the other hand, agent technologies allow proactive entities to operate autonomously and to interact with each other in order to accomplish a task. In particular, protocol languages such as the Lightweight Coordination Calculus (LCC) [94], MAP [119] and RASA [81] have been specified to express executable models of interactions within multi-agent systems. However, to put them into practical use, research needs to be carried out along the following directions:

- the establishment of frameworks that allows heterogeneous agents¹ to coordinate via interaction protocols;
- the development of software systems that implement the aforementioned frameworks;
- the exploitation and the evaluation of such systems, to assess their capability to support coordination tasks in complex, dynamic scenarios such as those briefly outlined earlier.

In respect to the first point, the OpenKnowledge (OK) framework [96] has been proposed in the past few years; it is an interaction-oriented approach according to which interaction models specified in a protocol language (e.g, LCC) are shared first-class entities which enable a priori unknown agents to engage in complex coordination patterns.

In respect to the second point, the OpenKnowledge system [34] has been developed within the European project OpenKnowledge² as a distributed peer-to-peer³ infrastructure which realizes the OK framework. In particular, such system is adopted as the underlying communication infrastructure

¹In the rest of this thesis, the terms “computational entity”, “process”, “agent”, and “peer” will be used interchangeably.

²<http://www.openk.org/>

³In the rest of this thesis, we will give to “peer-to-peer” the short name “p2p”.

that enables peers to find and coordinate with each other by publishing, discovering and executing multi party conversational protocols specified in the LCC language.

The work of this thesis has indirectly contributed to the second direction; and has mainly contributed to the third direction by means of the approach described in the next section.

1.2 The proposed approach

With the aim to apply the above mentioned interaction-oriented frameworks in realistic contexts, the approach taken has been to: (1) first design LCC interaction protocols to model coordination tasks in a complex domain such as emergency response; (2) then, provide a simulation environment, where to validate such interaction-based scenarios; (3) finally, conduct an experimental evaluation, to test the functionality of the OK system, and to start exploring the efficacy of different information gathering strategies in emergency response contexts.

The research has thus been carried out along three main lines: (1) LCC protocol engineering; (2) simulation environment development; (3) experimental evaluation.

LCC protocol engineering the effective design of interaction models is crucial to the application in realistic contexts. Since interaction models shape coordination patterns occurring in well defined settings (e.g., e-response, e-health, etc.), it is essential to study the functional requirements of the specific domain. This helps to identify some common interaction patterns occurring in emergency situations. Such interaction patterns in fact form the basis for identifying basic protocols which in turn constitute reusable interaction units needed to compose larger protocols in a modular way. We

designed interaction models starting from analyzing documents related to the current flood emergency plan foreseen by the Trentino region and from interviews with experts. Our analysis resulted in the modeling of the pre-alarm and the evacuation phases foreseen by the emergency plan in terms of LCC specifications.

It is worth to mention here that, such analysis affected the design of interaction models and provided requirements and feedbacks for the development of the OK system. Under this aspect, this thesis also contributes (though in an indirect way) to the development of the OK system itself.

Simulation environment development the developed simulation environment is composed of an agent network and a simulation engine. The first component models a hierarchy of e-response agents, which plan activities and coordinate with each other through predefined LCC interaction models; the second module simulates the environment where all the involved actors operate.

The developed simulator is used to: (1) informally validate interaction models, and investigate how appropriate LCC protocols are to reflect dynamic emergency coordination tasks; (2) evaluate the capability of the whole OK infrastructure to support different models of information sharing ; (3) assess, under specific assumptions, the efficacy of different information gathering strategies during e-response situations.

Experimental evaluation an experimental testbed has been designed, to test the OK infrastructure in a realistic and demanding case study, and to provide a preliminary evaluation on how different information gathering strategies (i.e., centralized and decentralized) impact emergency response tasks.

Specifically, simulations have been served as functional tests to determine

in which stage of the coordination mechanism the OK infrastructure failed in enacting the multiple interleaved interactions; this was done without entering in the details of the underlying communication infrastructure, thus, by considering the OK infrastructure as a black-box.

To get insights into the domain under study, and in particular, to evaluate how the above mentioned information-gathering strategies impact emergency response activities, we designed and conducted experiments. The result obtained show that our simulations adhere to realistic scenarios and that - under ideal conditions - such strategies are comparable.

In summary, this thesis contributes in the following four aspects:

1. The design of a set of interaction models written in the LCC protocol language to model complex coordination tasks in a realistic scenario such as a flood disaster;
2. The development of a simulation environment - built on top of the OK platform - to informally validate the just mentioned suite of interaction models;
3. A functional testing of the OK platform by means of the same simulator;
4. A preliminary evaluation on how different information gathering strategies impact emergency response tasks.

To the best of our knowledge, there is no prior work on the design and testing of complex interaction protocols in realistic contexts such as disaster situations. Also, while the OK platform has been exploited and tested only with a small number of simple interaction models involving mainly reactive web services [1], this thesis work focused on the exploitation and testing of the OK platform with multiple interleaved interactions between more proactive agents [111]. Finally, by providing a simulation

environment, this thesis takes a first step towards the assessment of how different information gathering strategies (e.g., centralized or distributed) affect emergency response efficiency.

1.3 Thesis structure

This thesis is divided in five parts.

Part I overviews state-of-the-art research in coordination techniques, in two different fields. In particular, Chapter 2 describes the dominant composition and collaboration techniques in the Web service area; Chapter 3 deals with the key concepts related to multi-agent interaction techniques and protocols, with an emphasis on the LCC protocol language.

Part II presents OpenKnowledge as a suitable framework to enact choreography-based interaction models. Chapter 4 introduces the core principles upon which the framework is based; Chapter 5 describes the main components of the OpenKnowledge system, and compares it with the dominant technologies.

Part III describes the design of LCC interaction models in specific emergency response scenarios and presents the environment realized to simulate them. In particular, Chapter 6 first introduces “crisis response” as the application domain of the proposed approach, and then presents the specific case study of a flood disaster; Chapter 7 describes the process that lead from the analysis of the scenarios to the design of LCC protocols and, consequently, to the requirements suggested for the development of the OK system. Finally, Chapter 8 describes the architecture of the simulation environment.

Part IV illustrates the experimental testbed used to: (1) provide a preliminary evaluation on how centralized and decentralized information gathering strategies impact emergency response situations; and (2) test the

overall functioning of the underlying supporting infrastructure, i.e., the OK system. Specifically, Chapter 9 first gives an overview of the experimental testbed; then it describes the hypothesis to be tested and the design of the experiments conducted to evaluate the information-gathering strategies; finally, it presents the results from the conducted experiments for both the evaluation of the strategies and the testing of the OK infrastructure.

Finally, Part V concludes the thesis with final discussions: Chapter 10 gives a summary of related work in the area of emergency response systems; finally, Chapters 11 and 12 draws conclusions and outline future work directions, respectively.

Part I

State of the Art

Chapter 2

Service-Oriented Coordination

While the Service-Oriented Architecture paradigm is being commonly accepted as a means to build distributed systems that deliver application functionalities as services, web services are emerging as suitable standards that provide an implementation of such paradigm. They can be composed, discovered and invoked thus allowing different organizations to provide and use service functionalities.

Since they are self-contained and self-describing XML-based components which enable interoperability of heterogeneous applications, they constitute the starting point for coordinating distributed processes. Web services are, in fact, sufficient for basic interaction needs but, by themselves, inadequate to integrate processes involving multiple participants. For this reason, languages and models are needed to provide coordination mechanisms fulfilling the execution of more complex interactions. Many proposals are available and some became already standards. The most important ones are: BPEL4WS (or BPEL) [4], WSCI [5], BPML [110], WS-CDL [99], OWL-S [72], WSMO [97]. There is no common agreement on the classification model for such languages but generally they are associated to orchestration and choreography languages [98, 87], or to web service composition techniques, which in turn are divided in static (e.g., BPEL,

WS-CDL) and dynamic (e.g., OWL-S) techniques [23]. Moreover, Kang et al. [63] distinguish between composition and collaboration techniques which adopt web services as basic and reusable software components. Table 2.1, taken from [63], shows the distinctive characteristics of both techniques taking WS-CDL and BPEL as examples of collaboration and composition techniques, respectively.

WS-CDL	BPEL4WS
The Collaboration layer specification	The Composition layer specification
Information driven	Explicitly invoking
Among participants	Within one single participant
Distributed controlling	Centralized controlling
Distributed controlling	Centralized controlling
Description language	Process execution language
Peer-to-peer	Centralized executor
Dynamic topology support	-
Candidate recommendation	Officially published
No commercial implementation	Websphere, ActiveBPEL, etc

Table 2.1: Collaboration vs. Composition

From the table above, while composition techniques (*a*) coordinate available services by means of a central controller that is fully responsible for the execution of the process and (*b*) focus on the modelling on one single participant, the counterpart collaboration techniques model interactions among participants in decentralized settings, where the control is distributed. This recalls the difference between orchestration and choreography approaches and take us to contrast some of the concepts outlined in the works mentioned so far: “programming in the large”, “coordinated interaction”, “collaboration”, “choreography” versus “programming in the small”, “coordinable”, “composition”, “orchestration”. In the next sections we follow this distinction and give an overview of both composition

and collaboration techniques¹ .

2.1 Composition techniques

The aim of composition techniques is to provide an extension of the Web service technology in order to support the creation of composite services out of simpler ones. The main methods are classified into two broad categories: workflow-based methods (e.g., BPEL, BPML) and methods based on Semantic Web techniques and AI planning for dynamic composition [90].

2.1.1 Workflow-based composition

In this kind of composition, the flow of the information and the invocation between services is known in advance. BPEL4WS [4] can be employed to define either abstract or executable business processes. BPEL abstract processes are not executable per se, but they can indirectly impose behavior compliance upon private processes executed by the BPEL orchestration controller. How? By defining sequencing rules on invoked operations upon the BPEL private process. In this way the BPEL orchestration engine can use the abstract process to validate and assure public protocol conformance of executing processes. BPEL is also employed to describe executable business processes. In this case, the process models the internal, actual behaviour of one participant; it does so by encoding a precise sequence of messages that are exchanged between partners and described by their WSDL definitions. Moreover, an executable business process is said to be “compositionally complete”, that is, exportable as a Web service eligible to participate in other compositions.

¹In the rest of this thesis, the terms “composition” and “collaboration” will be used interchangeably with the terms “orchestration” and “choreography” respectively.

2.1.2 Dynamic composition

The technique described in the previous section requires human activities since services have to be directly specified and protocols (or models) have to be manually defined at design time. This prevents complex services to be created dynamically, at runtime. Methods based on Artificial Intelligence aim at either generating the process model automatically or locate the correct services if an abstract process model is given.

Such an abstract process model can be provided by an OWL-S service. OWL-S [72] is one of the two most prominent frameworks for enabling automated Web service composition. It provides a service ontology that supplements the WSDL description of a Web service with semantics. An OWL-S service is defined by three parts:

- Profile: describes what the service does;
- Process Model: tells how the service works;
- Grounding: contains details on how to access the service

The idea is that automatic composition can be achieved by reasoning on these components. WSMO is the alternative framework to OWL -S: it goes beyond the definition of a service ontology and provides a complete methodology for constructing Semantic Web applications.

While frameworks like OWL-S and WSMO aim to enrich web services with semantic annotation, thus facilitating automated coordination of services (e.g., composition, discovery, invocation), AI planning techniques provide the means to create a plan representing the composite service. Some of these methods exploit semantic service descriptions to produce the plans. The work in [107] provides such an example: OWL-S Web service descriptions are translated in the knowledge domain of a planning system based on a Hierarchical Task Network (HTN). In [18], the authors introduce a

general architecture for service delivery and coordination and describe a composition component which provides flexible semantic service composition by exploiting OWL-S descriptions and Xplan, a planner partially based on HTN. In these works, the key idea is to sequentially compose the available web services, which are regarded as black boxes. In [89], a different technique based on “planning as model checking” is described to automatically compose BPEL web services: the code representing an internal business process is synthesized automatically starting from abstract BPEL specifications and a given business goal. This last method, in contrast to the others, has the advantage to deal with situations where external partner services are unpredictable and opaque in their internal status; however, the process of creating the plan is time-consuming. Although many other AI planning techniques exist for automating web service composition, it is not our aim to provide a complete list in the framework of this thesis.

2.2 Collaboration techniques

As previously mentioned, the aim of collaboration techniques is that of providing a global view of how the participants interact in order to achieve a common goal. Here, the perspective is switched, from the “viewpoint” of the single actor (orchestration) to that of all the involved actors, which, in this case, are treated equally (choreography). A choreography model does not describe any internal action (e.g., internal computation, data transformation) occurring within a participating service that is not directly related to an externally visible effect. In short, choreography foresees all the interactions between the participating services that are relevant with respect to the choreography’s goal.

Compared to composition techniques, collaboration techniques are less investigated and choreography languages are defined at a preliminary stage.

In the next section we give an overview of the main choreography languages.

2.2.1 WSCI

The Web Service Choreography Interface (WSCI) [5] was the first XML-based language attempting to provide a standard for describing message exchange among collaborating parties. However, WSCI is classified as a behavioural interface rather than a choreography language, since it describes the flow of messages exchanged by a web service participating in choreographed interactions with other services. WSCI specifies an abstract business process that is observable by external services. Therefore, like orchestration, focuses on the perspective of the single participant. Like choreographies, it does not describe internal tasks (e.g., internal data transformations).

2.2.2 WS-CDL

The Web Services Choreography Description Language (WS-CDL) [99] is the standard proposed by W3C in December 2004. It is an XML-based, abstract business process specification which defines interoperable, long-lived, p2p collaborations between web services. It introduces, among others, entities such as:

- *Role*: describes the external behavior that must be exhibited by a party to collaborate with other parties;
- *Relationship*: represents the mutual commitments needed between two parties to interact effectively;
- *Choreography*: defines the collaborations between interacting parties;
- *Interaction Activity*: determines an exchange of information between parties.

WS-CDL defines multi-party contracts that can be used in two ways: (1) to generate the behavioural interfaces needed by peers to participate at the collaboration and (2) to check whether a specific service - provided with its behavioural interface - is able to play a given role in that contract. Criticisms directed to the language regard, among the others: the lack of a comprehensive formal grounding, the lack of direct support for multi-transmission interactions, the lack of separation between the meta-model and syntax [12]. Moreover, there is no graphical support and only few implementations are available [38]. In this last respect, Kang et al. proposed WS-CLD+ as an extension of the WS-CDL language [62] and a first execution engine [63] for its enactment. Other implementations include pi4soa², a partial implementation presented in [49] and an agent-oriented solution [126].

2.2.3 Let's Dance

Let's Dance is a visual language introduced by Zaha and colleagues to model behavioral dependencies between service interactions [125]. The language is meant to be used during the design phase of a system, it supports the specification of both global and local models, allows to specify interaction models at different levels of abstraction, and to compose them. It has proven to be compliant with all the Service Interaction Patterns (SIP) [13], a set of 13 recurrent interaction scenarios used as a benchmark to assess the languages for service behavior modeling. Maestro has been developed as a tool to support the static analysis of global models, the generation of local models from global ones, and the interactive simulation of both local and global models [36].

²<http://sourceforge.net/projects/pi4soa/> [01/11/2010]

2.2.4 BPEL4Chor

BPEL4Chor [37] is a language built on top of BPEL, to extend orchestration models with choreographies. In BPEL4Chor, the core elements are: (i) *Participant behavior descriptions* define the control flow dependencies between activities, in particular between communication activities, at a given participant; (ii) A *participant topology* determines the structural aspects of a choreography by specifying participant types, participant references, and message links; (iii) *Participant groundings* describe the technical configuration of the choreography by pointing to actual links to WSDL definitions, and establishing XSD types. It complies with almost all service interaction patterns, promotes reusability since technical details are confined to the groundings layer, and directly supports multi-party interactions. Regarding the supporting tools, a web-based editor³ exists that enables designers to create choreography models in a graphical way.

³<http://www.bpel4chor.org/editor/> [01/11/2010].

Chapter 3

Agent-Oriented Coordination

As presented in the previous chapter, the SOA framework has the goal to provide loose coupling among interacting software agents which are distributed over a network and created by different organizations, with different platforms. The similarity with Multi-Agent Systems (MASs) becomes apparent: a MAS is, in fact, a loosely coupled network of intelligent agents that interact to solve problems that are unsolvable by the individual capacities or knowledge of the single entity. However, there is a fundamental difference between SOA and MAS that lies in the definition of an agent. A “SOA agent” is a software agent, that is, a program that implements and accesses a web service. In a MAS, the agent is an intelligent agent, that is, an independent entity which is able to make decisions autonomously. While software agents are involved in stateless communications, intelligent agents can engage in complex interactions by means of sophisticated communication and coordination mechanisms and can easily adapt to changes in the environment they inhabit. This makes agent techniques suitable to complement and enhance the more inflexible nature of Web service technology. In the next sections, we give an overview of the main agent communication and coordination techniques and sketch some approaches which use these techniques to strength web service technology.

3.1 Agent communication languages

An Agent Communication Language (ACL) is a mechanism allowing agents to exchange information and knowledge. Existing ACLs for MASs are KQML[45] and FIPA-ACL [47]. Even though the latter has superseded the former, the principles underlying the two are similar. They are based on the speech act theory [8, 102], according to which verbalizing certain utterances (performative speech acts) means to “do things” or “perform”; a message therefore specifies a performative (or communicative act) which tells whether it is an assertion, a command, a query or any other element of a predefined set of performatives. A KQML message is organized in three layers:

- Content Layer: it represents the actual content of the message encoded in a given representation language. Different representation languages are allowed, including languages expressed as ASCII strings and those expressed using binary notation;
- Communication Layer: it specifies lower-level communication parameters, such as the identity of the sender and the receiver, and a unique identifier for the communication;
- Message Layer: it is the fundamental layer which determines the type of performative; it also specifies the content language, the ontology used and some type of description of the content.

As stated above, FIPA-ACL is conceptually similar to KQML. Both ACLs are specified with a formal semantics which is intended to provide the meaning of the communicative acts: each message comes with a set of (feasibility) preconditions and postconditions (rational effects). The former represent the necessary conditions that must hold before a message is sent and the latter determine the state in which the sending agent must be in

after the message delivery; usually they also specify the conditions that must be satisfied in the recipient agent.

For what concerns the integration of FIPA's technologies with service-oriented ones, [18] sketches an agent-based approach where FIPA-ACL standards adopted in JADE are combined with Semantic Web Services (OWL-S) to realize a flexible coordination infrastructure, able to operate in highly dynamic settings. P. A. Buhler et al. [25] recognize the benefit of using MAS techniques for flexible enactment of enterprise workflows and, starting from a comparison between BPEL and the FIPA-IP standard, suggest the use of a BPEL specification to express the initial social order of a MAS.

In [67], the authors analyse the two ACLs, compare them and explain their limitations: it is up to the agent programmer to conform to the semantics defined in the specification. As a consequence, the problem is to pass from the theory (formal semantics) to the code and the process is left to the intuition of the designer. This clearly leads to problems when coordinating groups of heterogeneous computational entities. However, since compliance with the formal model is not enforced, ACL standards are adopted in some multi-agent platforms such as JADE [17].

Besides the problem of relying on agent designers to avoid messages being misinterpreted, a crucial issue of ACLs is that while they guarantee knowledge sharing, effective coordination is not ensured. Cost et al. [31] well identify these two distinct and separate problems which interacting agents face when acting in open and dynamic environments: while the knowledge sharing problem between agents is solved with shared ontologies and translation of their respective representation languages, coordination requires agents to reason about the next action to perform (e.g., the message to send), the anticipated actions of others and the environment; in this way, coherent interactions aimed at completing a common task can take

place. In short, agents usually engage in complex interactions which use ACL performatives as building blocks. Another perspective of this problem is provided by Greaves et al. [54] which identify the *Basic Problem*:

“Modern ACLs, especially those based on logic, are frequently powerful enough to encompass several different semantically coherent ways to achieve the same communicative goal, and inversely, also powerful enough to achieve several different communicative goals with the same ACL message”.

Basically, it is unfeasible for an agent, whose aim is to select the next action to take in an interaction, to reliably deduce the intentions and goals implied in the use of an ACL performative by another agent; therefore, since it lacks the more rich context of the interaction, it comes to search in a vast space of possible actions. As Maudet et al. [75] observe, the complexity of this task is made worse by the difficulty of accessing an agent’s mental-state (especially in open environments) and by the fact that not always agents are “sincere”, hence trustable.

3.2 Conversation policies

To overcome the limitations of ACLs, vast research has been conducted. The concept of “conversation policy” as a preplanned pattern of message exchange that two or more agents agree to comply with has been introduced. A conversation policy (or protocol) narrows the possible choices an agent has to continue its dialogue with its counterparts; it establishes the allowed sequences of semantically coherent messages exchanged to achieve a goal by constraining them. As Maudet et al. [75] observe, a conversation provides the context and has its own semantics and this coincides with the semantics of an ACL, which thus cannot be reduced to the sum of the single performatives’ semantics.

Conversation policies as defined by Greaves et al. [54] are conceived as abstract sets of fine-grained constraints, each handling a certain aspect of the conversation (i.e., uptake acknowledgment, synchrony, exception handling, etc.). They should be shared among the agents and enforced by the computational models adopted by each of them.

In order to model conversation policies on an abstract level, extensions of the Unified Modelling Language (UML) have been proposed, such as the AUML formalism [84, 85]. It allows to represent agent interactions by means of collaboration, sequence and activity diagrams. These kind of protocol diagrams are used by FIPA to provide an Interaction Protocol (IP) standard, i.e., a set of high-level interaction protocols (e.g., auction, contract-net, negotiation) [46]. Tools have been proposed for mapping high-level AUML diagrams into Petri nets, thus to generate code structures from the drawings [26].

Computational models used to implement conversation protocols range from Finite State Automata (FSM), to Pushdown Automata (PDA), to Petri Nets and Colored Petri Nets. For example, Iwao et al. [59] specify conversation policies by means of FSM: so called policy packages separating the agent's inner state from the state transitions and the rules for state transitions are dynamically exchanged among agents which are required to adopt the same architecture in order to interpret a policy package. Martin et al. [73] also use an automata (specifically a PDA) for protocol representation; however, they provide mechanisms to statically and dynamically upgrade it. Petri Nets are preferred by de Silva et al. [35] since they allow to parallelize conversation (e.g., messages sent in multicast). The advantages of using Colored Petri Nets (CPNs) to model conversational protocols are presented in detail by Nowostawski et al. [83] which underline the ability of such computational models to model, beside concurrency, the different participants' roles; and to allow the reuse of conversation structures.

Further research (among many others) on the use of CPNs for protocol representation has been conducted by Cost et al. [29, 30, 31] and Lin et al. [68].

3.3 Social agency

Alternative approaches to conversation policies have emerged which consider agents as members of an artificial society. Such society-based approaches underline the importance to explicitly define societal norms in an interaction among agents [105, 104]. Societal norms represent constraints, that is, restrictions an agent may choose to adhere to, in order to be part of a society and thus gain benefit from it (e.g., from services, knowledge provided by other agents). In particular, concepts as “commitments”, “obligations”, “power” (meaning hierarchical relations) have been introduced to model such artificial social systems. The following sections present briefly, and not exhaustively, some of the most relevant approaches. More comprehensive surveys tackle the issue of specifying agent coordination under different perspectives: Artikis et al. [6] provide a definition of Open Agent Systems (OAS) through the review and comparison of four frameworks (i.e., normative systems, enterprise modeling, commitment protocols and Electronic Institutions) highlighting their ability to model normative relations; McGinnis et al. [76] discuss only those approaches that can be ascribed to “first-class protocols” by stressing the feature of being dynamically composable.

3.3.1 Commitment protocols

The concept of social commitment has been used by Yolum and Singh [123] to develop a framework based on *commitment machines*. Here social norms are represented by commitments. A commitment $c(x, y, q, p)$ means that

the debtor x is committed to the creditor y to bring about the condition p when the condition q is satisfied. There can be non-conditional (or base-level) commitments (i.e., q is true) and meta-commitments (i.e., p involves other commitments). Commitments provide meaning to a protocol and have operations defined on them (e.g., create, discharge, cancel, etc.) which are executed as a result of a message sent or received. Agents are supplied with inference mechanisms that allow them to reason about the action to be taken. The works in [124, 39, 28, 40] present and describe the use of commitment protocols.

3.3.2 Normative systems

In normative systems, the approach is similar to that of commitment machines but social norms are represented by legal and illegal actions instead of commitments. As before, agents have the proper machinery to create and manipulate norms when messages are sent or received. The work in [7] presents this kind of framework.

3.3.3 Dialogue-game protocols

The approaches that lead to these kind of protocols were inspired by the theory of Walton and Krabbe [122], according to which dialogues can be categorized into six types (e.g., persuasion, information-seeking, negotiation, etc.) depending on the individual and joint goals of the participants, and on the information and the aptitude (e.g., being cooperative, competitive) they initially have. Research has been carried out to formalize the theory and to apply it to enable inter-agent communication [91].

Maudet et al. [75] offer a broad survey of both commitment-based and dialogue-game protocols.

3.3.4 Electronic Institutions

Electronic Institutions (EI) [44, 42, 43, 50] provide a framework to specify artificial social systems where agents are supposed to follow social norms to engage in interactions. The core components of an EI are [42]:

- Roles: they represent the agent capabilities, that is, the actions allowed when an agent takes a given role, and are organized in hierarchies to specify, for example, subsumption and exclusivity between roles;
- Dialogic framework: it determines the message format of communicative acts, i.e., the ontology, the content language, a list of illocutions (type of performatives), roles and their relationships. The dialogic framework hence permits agents to interact meaningfully.
- Scenes: a scene defines a specific task, i.e. a structured exchange of messages between roles, within the whole interaction (e.g., negotiation). Scenes are defined as finite state machines and are connected together, thus forming a so called performative structure. Agents only interact directly within a scene, move simultaneously between states, can take different roles, and can pass from one scene to another depending on normative rules (e.g., obligations and commitments), these being defined and administered externally by a central process.
- Performative structure: a network of scenes describing how agents can move between scenes, provided that they satisfy certain rules which are dependent on the roles they occupy and the performative they execute.
- Normative rules: a normative rule expresses obligations or prohibitions on agent behavior and is defined by an antecedent (a list of

scene-illocution pairs) and a consequent (the predicates that must hold if the antecedent illocutions have taken place).

Tools assisting the design of EIs exist: ISLANDER is such an example [44]. It provides a graphical user interface that allows users to edit performative structures, scenes and illocutions. Another tool offering computational support for EI specifications is the one presented in [50]; it is a rule-based system for executing a set of normative rules which are used to derive the permissions and obligations of the agents in each state.

EIs offer the advantage that agents involved in interactions will actually conform to the established norms, this resulting in reliable coordination mechanisms. However, some drawbacks cannot be avoided: (1) since a finite-state model is adopted, all the possible conversations are shaped before they actually occur, thus limiting interaction flexibility; (2) since an administrative agent synchronizes the agents, the objective to build decentralized systems is not fulfilled.

3.3.5 Lightweight Coordination Calculus (LCC)

Alternative approaches exist that overcome the limitations of the EI framework. These are based on the use of protocol languages: executable specifications which are used directly to build real multiagent systems [119]. The feature of being inspectable/executable protocols avoid to translate abstract specifications into computational models.

The Lightweight Coordination Calculus (LCC) is a protocol language according to the above definition. It is based on logic programming [94] and is used to describe interactions among distributed processes, e.g., agents, web services [92, 93]. LCC was designed specifically for expressing p2p style interactions within multi-agent systems, i.e., without any central control; henceforth, it is well suited for modeling coordination of software compo-

nents running in an open environment. Its main characteristics are the flexibility, the modularity and the neutrality to the underlying communication infrastructure. It supports, on one side, the independence and autonomy of the peers and, on the other side, the collaborative process essential to achieve a common goal in a distributed-knowledge and dynamic environment.

LCC syntax and semantics

The formal syntax of LCC is presented in Figure 3.1.

$$\begin{aligned}
 \textit{Framework} & := \{ \textit{Clause}, \dots \} \\
 \textit{Clause} & := \textit{Role} :: \textit{Dn} \\
 \textit{Agent} & := a(\textit{Type}, \textit{Id}) \\
 \textit{Dn} & := \textit{Agent} \mid \textit{Message} \mid \textit{Dn then Dn} \mid \textit{Dn or Dn} \mid \textit{Dn par Dn} \mid \textit{null} \leftarrow \textit{C} \\
 \textit{Message} & := M \Rightarrow \textit{Agent} \mid M \Rightarrow \textit{Agent} \leftarrow C \mid M \Leftarrow \textit{Agent} \mid C \leftarrow M \Leftarrow \textit{Agent} \\
 \textit{C} & := \textit{Term} \mid C \wedge C \mid C \vee C \\
 \textit{Type} & := \textit{Term} \\
 \textit{M} & := \textit{Term}
 \end{aligned}$$

Where *null* denotes an event which does not involve message passing; *Term* is a structured term and *Id* is either a variable or a unique identifier for the agent.

Figure 3.1: LCC formal syntax

An LCC interaction model is a set of clauses, each of which defines how a role in the interaction must be performed. Roles are described by their type and by an identifier for the individual peer undertaking that role. Participants in an interaction take their *entry-role* and follow the unfolding of the clause specified using a combination of the sequence operator (“*then*”) or

choice operator (“*or*”) to connect messages and changes of role. Messages are either outgoing to (\Rightarrow) or incoming from (\Leftarrow) another participant in a given role. During an interaction, a participant can take on more than one role, and can recursively take the same role (for example when processing a list). Message input/output or change of role is controlled by constraints defined using the normal logical operators for conjunction and disjunction.

LCC example

A basic LCC interaction is shown in Figure 3.2.

$$\begin{aligned}
 a(r1, A1) &:: \\
 &ask(X) \Rightarrow a(r2, A2) \leftarrow need(X) \text{ then} \\
 &update(X) \leftarrow return(X) \Leftarrow a(r2, A2) \\
 \\
 a(r2, A2) &:: \\
 &ask(X) \Leftarrow a(r1, A1) \text{ then} \\
 &return(X) \Rightarrow a(r1, A1) \leftarrow get(X)
 \end{aligned}$$

Figure 3.2: LCC example: double arrows (\Rightarrow, \Leftarrow) indicate message passing, single arrow (\leftarrow) indicates constraint satisfaction.

The peer identified by the value of the variable $A1$ playing the role $r1$ verifies if it needs the info X (pre-condition $need(X)$); if it does, $A1$ asks the peer identified by the value of the variable $A2$ for X by sending the message $ask(X)$. $A2$ receives the message $ask(X)$ from $A1$ and then obtains the info X (pre-condition $get(X)$) before sending back a reply to $A1$ through the message $return(X)$. After having received the message $return(X)$, $A1$ updates its knowledge (post-condition $update(X)$).

The constraints embedded into the protocol express its semantics and

could be written as first-order logic predicates (e.g., in Prolog) as well as methods in an object-oriented language (e.g., in Java). Furthermore, these constraints could hide simple functionalities (i.e., provided by web services) as well as very complex AI algorithm. This is the characteristic of modularity previously mentioned that allows to separate the protocol from the agent or service engineering.

While performing the protocol, peers can therefore exchange messages, satisfy constraints before/after messages are sent/received and jump from one role to another so that a flexible interaction mechanism is enabled still following a structured policy, which is absolutely necessary for team-execution of coordinated tasks.

Coordination in LCC

An agent willing to coordinate via LCC protocols must have a way to receive the protocol, unpack it, interpret what are the actions to perform next and upgrade the state of the conversation. This is done through a *protocol expansion* mechanism which consists of:

- Extracting the protocol embedded in the message received;
- Extracting from the protocol the clause pertaining to its role and identifying its part of the conversation;
- Applying rewriting rules to compute a new clause (that will be substituted to the old one in order to produce a new protocol), a set of outgoing messages, and a set of unprocessed messages;
- Sending the outgoing messages, each containing a copy of the new protocol.

The above steps and the rewriting rules are described in detail in [94]. To enact the above-sketches coordination mechanism, the following compo-

nents need to be present in an agent process: (i) a message encoder/decoder for sending and receiving messages through the communication infrastructure chosen to transmit messages on the wire; (ii) a protocol expander performing the just mentioned steps; and (iii) a constraint solver used to satisfy the pre- and post-conditions specified in the protocol.

The fact that the protocol can be exchanged between agents while the conversation evolves, allows for flexibility and adaptivity. On one hand, LCC protocols can be designed in a way that confine agents to be mere executors of decision procedures; in this case, the protocol is fully constrained and imposes strict “social norms” which limit the autonomy of the agent; on the other hand, they can be very flexible, for example by just declaring the messages allowed to be exchanged; in this second case, agents are required to apply reasoning techniques in order to compute the next actions to be taken. They could even modify the protocol so that the one sent next differs from the one received. McGinnis et al. [77, 78, 79] conducted research on how to make protocol transformation in a way that still guarantees the continuation of meaningful dialogues.

Although the LCC protocol provides flexible specifications, some drawbacks were discovered: Besana et al. [19] present these limitations and extend the language so to overcome them. In particular, they introduce two operators: the *scene* operator which allows to model abstractions of interaction models, thus to compose them in a more straightforward way; and the *parallel* operator that permits to create new processes performing roles at run-time.

As far as composability is concerned, the RASA executable protocol language was designed where dynamic composition is intrinsic to the language. RASA combines constraints and process algebra to model interaction protocols as first-class entities [81]. As for LCC, the agent process is decoupled from the protocol but the ability of participants to compose protocols at

runtime to engage in more complex interactions is a built-in feature. A comparison of the two protocols by analyzing different characteristics is provided by McGinnis et al. [76].

Integration with Web service standards

The benefits of integrating multiagent techniques with web service standards are pointed out in different research works [93, 119, 118, 27, 100, 55, 56].

Li Guo et al. [55, 56] apply techniques to enact business workflows expressed in BPEL4WS in a distributed manner, i.e., by using a multiagent platform where agents exchange LCC protocols. In [55], an LCC interaction model is directly derived from a BPEL4WS specification and passed around the agent network; in [56], the approach of using an LCC protocol to interpret a BPEL4WS model is preferred.

A protocol language similar to LCC, MAP, has proven to be effective in building multiagent systems within a Web service architecture, thus allowing complex communication patterns: Walton [119] describes a service approach, the key idea being that of building a MAS service composed by proxies in charge of communicating via interaction models. An agent is represented by one or more proxies and an external web service, which embodies its decision procedures (defined as constraints in the interaction models). The approach is substantiated in the MagentA agent system [120]. Here, the agent technique is adopted to enable web service composition by defining pre-prepared plans written in MAP. The same protocol is used for web service invocation [117]: it specifies the correct order in which the operations should be invoked as well as inter-argument dependencies, thus enhancing industry-standard WSDL specifications.

Another attempt to integrate agent techniques with the web service world is made by Giunchiglia et al. [52]: here LCC is extended with

contextual information in order to facilitate semantic matching at run-time, this being crucial for enabling automatic service composition; although interaction protocols are predefined workflows, the advantage is that web services are not assumed to be semantically annotated.

In relation to web service composition, other agent-oriented methods exist: for instance, Casella et al. [27] propose a tool for translating AUML visual diagrams into WS-BPEL and WSDL definitions. Also, a protocol-based approach is advocated by Singh et al. [106] to model and enact business processes. Here, the emphasis is on how the modular nature of protocols can be exploited to enable their abstraction, thus allowing protocols to be composed and reasoned about.

Recently, protocol languages such as LCC and MAP have been regarded more as executable choreography languages rather than logic-based workflows enabling service composition [22, 19, 9, 10, 11]. Barker et al. [10, 11] used them to enact e-science experiments where the transmission of huge amount of data benefits from a choreography model in terms of decreased data transfer and bottleneck likelihood. In particular, in [10]/[11], Open-Knowledge/MagentA is presented as the framework for the enactment of distributed LCC/MAP choreographies which are then compared with existing languages such as WS-CDL, Let's Dance and BPEL4Chor.

Finally, Miller et al. [82] propose the RASA language as a viable solution to coordinate web service and agent processes in e-science workflows.

Part II

An Interaction-Oriented Approach to Knowledge Sharing: OpenKnowledge

Chapter 4

The Framework

In the previous two chapters, we have given an overview of state-of-the-art research in coordination techniques in two different (but combinable) fields, namely the web service and the multiagent fields. In this chapter, we restrict our attention to OpenKnowledge, the framework developed within the European project OpenKnowledge [95], and around which our work revolves; we describe here its underlying principles.

As emerged from our state-of-the-art analysis, effective knowledge sharing and coordination between (reactive/proactive) processes across open environments, is made possible, respectively, by achieving semantic interoperability and by permitting computational entities to engage in complex interaction patterns. The OpenKnowledge framework we introduce in this chapter addresses both issues with few general principles.

The basic intuition is that by promoting interaction specifications as the currency of knowledge sharing, it is possible to have a context in which the knowledge that is to be exchanged may be interpreted [96]. In this way, semantic heterogeneity problems can be reduced; in addition, they can be solved with techniques which prevent different parties to agree on

a common conceptualization of the world a priori, i.e., before the actual interaction. Instead, semantic agreement applies only for the purpose of the interaction and within its scope.

According to the just described vision, it is fundamental to have interaction specifications fulfilling certain criteria.

The first criteria consists of specifications which guarantee the separation between the interaction protocol and the processes that will take part in it. For example, if the protocol is being used to compose a set of services, it should not specify which atomic services will be involved. This is crucial to *allow reuse of interaction specifications*, hence, to permit heterogeneous peers to make use of them.

The second criteria requires to devise specifications which are independent on the computational model adopted to run them. For example, a peer could run the interaction model locally and contact partner peers only when needed; or, it could opt for a distributed computation where each party is separately running its piece of model; or, finally, interactions could be passed around as scripts thus to produce complex interactions [96].

It is therefore crucial to *ensure neutrality towards the computational strategy adopted for executing the specifications*.

The OpenKnowledge framework promotes the use of the LCC language to specify models of interactions; the good reason for that lies in the fact that LCC represents an executable protocol language which fulfills the two criteria above, and is expressive enough to shape complex interaction patterns between peer roles. LCC has already been presented in Chapter 3, Section 3.3.5; we refer to a basic description of it in the Section 5.1 of the next chapter, where the core components of the OpenKnowledge system

are described.

Once an interaction model is specified, coordination between peers has to take place. In order to form teams for the purpose of completing a task effectively, three principles must be put in action [96]:

1. In open environments, peers should interact in a meaningful way, i.e., the *knowledge exchanged should be understood within the context of a given interaction*. This is accomplished by adopting dynamic ontology matching techniques that, at run-time, map terms in the interaction models and help to select peers which are not known in advance but are suitable to take part in a given interaction;
2. *Interaction models should be searchable and peers should be able to select suitable partners to interact with*. This is accomplished by relying on some distributed discovery service(s) which could use simple keyword-based techniques to retrieve appropriate interaction models; also, peers may select partners according to different strategies, one of which could be relying on their past experiences;
3. *Full automation is not a prerogative: humans might need to be on the loop*. The provision of visualization components is thus required. This is accomplished by allowing default visualization modules and/or modules fully customizable by both interaction models and peer designers.

In the next chapter we present the OpenKnowledge system, i.e., the infrastructure which has been developed to implement the framework described here.

Chapter 5

The OpenKnowledge System

The OpenKnowledge system has been developed to translate the principles previously described into a concrete piece of software. It provides the underlying p2p communication infrastructure needed to execute coordinated tasks between peers. As already mentioned, the key concepts in OpenKnowledge are [95]:

- The interactions between agents, defined by interaction models written in LCC and published by the authors on a peer-to-peer infrastructure with a keyword-based description;
- A distributed infrastructure, denoted as OpenKnowledge Kernel, that supports the publishing, discovery, execution, monitoring and management of the various interaction models.

In Section 5.1 we recall some LCC basics and, in Section 5.2, we describe the core ideas behind the OpenKnowledge Kernel, which allows the execution of models of interactions among agents.

5.1 LCC interaction models

Interaction models are specified in the LCC language [92], an executable choreography language based on process calculus, and published by the

authors on a distributed discovery service (DDS) with a keyword-based description [66]. LCC is designed [92] to model coordination of software components running in an open peer-to-peer environment (without a central management): it supports, on one side, the independence and autonomy of the peers and, on the other side, the collaborative process essential to achieve a common goal in a distributed-knowledge and dynamic environment.

As described in Section 3.3.5, an interaction model in LCC is a set of clauses, each of which describes the behaviour of an agent role. This is defined by structuring message exchanges with other roles and role changes; and by applying constraints defined using the normal logical operators for conjunction and disjunction. During an interaction, a participant can take on more than one role and can take on the same role recursively.

In OK, constraints have a key role in that they are bridges between the interaction and the peers' knowledge: they are used to query values or to update the local knowledge of a peer. The LCC specification does not enforce any particular constraint solver, hence, different engineers might choose different techniques: web services, object oriented language methods, first order logic predicates, human intervention, etc. The OK framework suggests to solve constraints by means of Java methods defined in so called *OpenKnowledge Components*. These components are described in more detail in Section 5.2.1.

Figure 5.1 shows an example of a simple LCC interaction. Here the participant *C*, playing the role *controller*, first tries to get the time (precondition *getTime(Time)*) and then, if this precondition is satisfied, it sends the message *request_info(Time)* to the participant *FS* playing the role *food_simulator*. When the flood simulator receives the message from the controller, it first resolves the *floodChanges(Time, Changes)* constraint, that calculates changes in water level at time *Time*, and then it sends back

a message with the changes ($flood_info(Changes)$) to the controller. The controller, after having received the $flood_info(Changes)$ message, updates its local knowledge about the flood with the postcondition constraint $updateFlood(Changes)$. At the end of the controller role definition, we also have an example of recursion ($a(controller, C)$).

```
a(controller, C) ::
  request_info(Time) ⇒ a(flood_simulator, FS) ← getTime(Time) then
  updateFlood(Changes) ← flood_info(Changes) ⇐ a(flood_simulator, FS) then
  a(controller, C)

a(flood_simulator, FS) ::
  request_info(Time) ⇐ a(controller, C) then
  flood_info(Changes) ⇒ a(controller, C) ← floodChanges(Time, Changes)
```

Figure 5.1: Simple LCC interaction.

5.2 The OpenKnowledge Kernel

The OpenKnowledge Kernel [34] uses p2p technology to provide a distributed infrastructure which enables assorted services and applications to interact according to predefined taskflows; it permits peers to publish, search and run interaction models as well as to share and download so called OpenKnowledge Components (OKCs), i.e., the software code providing the services needed to execute the taskflows.

Each peer willing to participate in an “OK-enabled” interaction has to install the OK Kernel whose architecture is sketched in Figure 5.2 [34].

A peer can manage OKCs and store interaction models¹ (IMs), i.e., formal specifications written in a suitable language (e.g., LCC). Each IM de-

¹In the rest of this thesis, we will give to “interaction model” the short name “IM”.

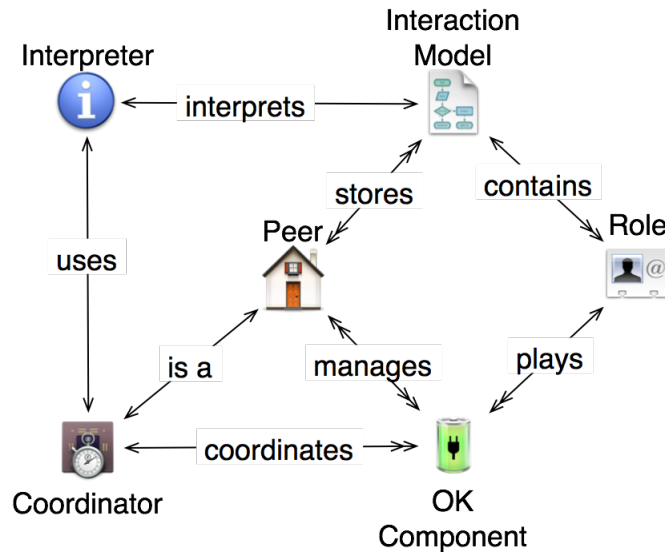


Figure 5.2: OK Kernel Architecture [34]

defines roles and the interactions between them. OKCs are able to fulfill one or more roles in a given IM, and are coordinated by means of a peer acting as a coordinator in the network. In order to be selected as a coordinator, a peer needs to include an interpreter able to parse the IM specification at hand. Notice that the OK Kernel infrastructure is not strictly tied to the language used to specify workflows. Consequently, different languages will require different interpreters.

The next sections describe in more detail the OpenKnowledge components (Section 5.2.1), the core modules of the kernel (Section 5.2.2) and the services it provides (Section 5.2.3).

5.2.1 OKC components

As mentioned in Section 5.1, the constraints defined in an IM can be solved by means of Java methods which are in turn specified within OKCs. They are software components with the following characteristics:

- Providing the bridge between IM's constraints and the peer's knowledge;
- Allowing a peer to play different roles in different IMs;
- Allowing modularity and reuse of code.

A peer willing to play a role in a given interaction, must be capable of solving the constraints defined for that role. For this purpose, it needs to add OKCs in its repository. An OKC is a Java class containing methods which can be used to solve constraints defined in IMs. When a constraint in an IM has to be solved, a Java method declared in an OKC “associated” to the peer, that plays the role containing this constraint, is called [20]. A peer knows whether it is capable of playing a given role by applying a matching algorithm during the subscription phase, which is described in more detail in Section 5.3.1.

When a peer takes part in an IM, new instances of the related OKCs are created and stored in a local repository. An OKC, indeed, can be used in more than one interaction but a single OKC instance has a scope only within the associated interaction. Therefore, while a peer can run different IMs at the same time, data cannot be shared among the different runs. However, the peer's knowledge updated within a given interaction might need to be accessed in the context of another interaction running in parallel. Therefore, there is a need for a mechanism allowing an OKC to access the peer's local knowledge, i.e., information that persists across multiple interactions.

For this purpose, the OK framework provides a *peer access mechanism* which uses Java annotations [20]. On one side, a peer exposes a set of methods which access its internal knowledge and which are semantically annotated as in Figure 5.3. On the other side, an OKC lists the peer's

methods it needs to use in order to access the peer local knowledge (Figure 5.4).

A peer eventually instantiates and stores the `PeerAccessor` class containing the above mentioned peer's methods. Before adding an OKC to its local repository, it matches the annotated signatures of the methods exposed by the peer and those of the peer's methods required by the OKC. This is done by transforming them into trees and verifying their distance [53, 51]. The result of this operation is the creation of an adaptor which allows an OKC to access the peer's local knowledge. The described mechanism is created through a manager module which acts as a mediator: it ensures decoupling between the peer and the OKCs. More details on the mediator module are given in Section 5.2.2. Figure 5.5 shows how peer's knowledge is accessed through the method `invokePeer('getMaxTimestep', arg)`, called within the OKC component.

```
public class PeerAccessor implements PeerAccess {
    .....
    @MethodSemantic(language="tag",
                    args={"max_timestep"})
    )
    public boolean getMaxTimestep(Argument MaxTimestep){
        MaxTimestep.setValue(this.maxTimestep);
        return true;
    }

    @MethodSemantic(language="tag",
                    args={"max_timestep"})
    )
    public boolean setMaxTimestep(Argument MaxTimestep){
        this.maxTimestep = (Integer) MaxTimestep.getValue();
        return true;
    }
    .....
}
```

Figure 5.3: Annotated peer's methods

```
@RequiredPeerAccess(  
    methods={"getMaxTimestep(max_timestep)",  
            "setMaxTimestep(max_timestep)",  
            "getTimestep(timestep)",  
            ....}  
)
```

Figure 5.4: Peer's methods required by an OKC

```
public int getCycles(){  
    ....  
    try {  
        Argument arg = new ArgumentImpl("max_timestep");  
        invokePeer("getMaxTimestep", arg);  
        maxtimestep = (Integer) arg.getValue();  
    } catch (AdaptorException e) {}  
    ....  
}
```

Figure 5.5: OKC: access to peer's local knowledge

It is important to notice here that from within OKC methods, new interactions can be enacted. This can be done *implicitly* (e.g., it is up to the peer to decide whether to enact a new interaction) or *explicitly* (e.g., the method enforces the enactment of a new interaction) [20]. This is an important feature that will be heavily exploited in our simulation.

An OKC can be shared across the network by wrapping it into a JAR archive which is then published on a distribute discovery service via a dedicated user interface. More details on this are to be found in Sections 5.2.2 and 5.2.3.

5.2.2 Kernel modules

The core modules included in the OK Kernel are depicted in Figure 5.6 and are briefly described below [34]:

Component Repository: it allows the local storage of OKCs implemented by the peer or downloaded from the network. Retrieval of OKCs is

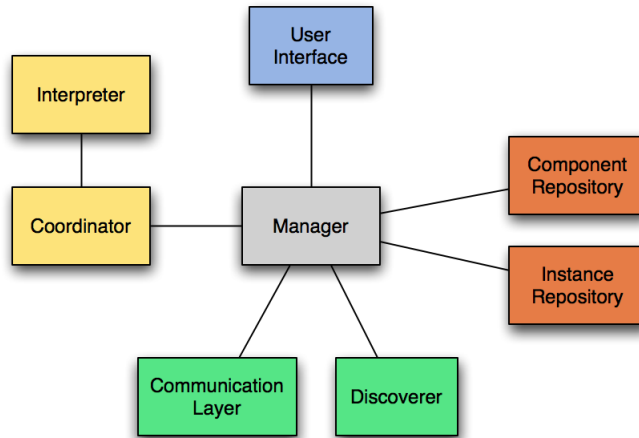


Figure 5.6: Peer Modules [34]

also possible and is achieved by means of keyword-based mechanisms; users can provide labels for OKCs via a predefined user interface;

Instance Repository: since an OKC can be used in more than one interaction, an OKC instance has to be created at each run. A peer, thus, manages the created instances by means of this repository.

Communication Layer: it sends asynchronous messages to other OK peers using the TCP/IP transport protocol. Messages contain the sender and receivers IP address, port and identifier (that is unique inside the peer) and the information being delivered [33];

Control Manager: this module ties together all the other modules which can thus communicate in a decoupled way. There is no direct communication between modules except the one with the control manager. This engineering choice allows to reduce the dependences between objects;

Coordinator: it enables a peer to play the coordinator role. It executes the IM locally, interprets it, maintains the state of the interaction and

simulates message exchanges; when it encounters a constraint to be solved, it calls the peer having the OKCs playing the IM roles where the constraint is defined. The coordinator is key to the execution of an interaction model; more details on the interaction lifecycle are given in Section 5.3.

Interpreter: it parses an IM, applies the rules and identifies the constraints that have to be solved by OKCs. It is the module that allows a peer to execute an IM and, more generally, that permits the decoupling between IMs and OKCs;

User Interface: it provides an user interface (i) to publish, search and subscribe to IMs, (ii) to public and download OKCs and (iii) to visualize interaction state and constraint satisfaction.

5.2.3 Kernel services

The OpenKnowledge Kernel comes with a set of three predefined services. One of such services, the *distributed discovery service* (DDS), is crucial for the enactment of an interaction; the other two are optional but make peer interactions more effective. These kernel services are described below:

Distributed Discovery Service (DDS): it provides a common distributed storage for IMs and OKCs [65]. It is also responsible for the subscription of a peer to a role and for the subscription of the coordinator. Finally, the Discovery Service can be completely distributed: this means that there can be more than one instance of the service running at the same time on different computers.

The DDS algorithm [65] takes into consideration the frequency of term occurrences in descriptions: most popular terms are found, with a high

probability, by a random search across the peers in the network, while rare terms are found by exploiting standard DHT principles.

Ontology Matching Service: it allows to run interactions between participants without a semantic agreement established a priori. Within the OK system, two types of matching problems are tackled. The first one regards the matching between the peer's query, which expresses the desired service functionality, and the descriptions associated to candidate IMs suitable for the requester. Currently, this problem is addressed by the DDS by matching IM descriptions using a simple query expansion mechanism.

The second one is related to the mapping between the OKC Java methods, which express the peer's capabilities, to the constraints in the IMs. The type of matching adopted [53] produces a confidence level parameter that reflects the overall similarity between the constraints and the methods, that is, how well the peer can execute the interaction;

Trust Service: it helps a peer in the choice of partner peers for a given interaction. It is used during interaction bootstrap (see Section 5.3.2 for more details): a peer, after having chosen a suitable IM from a list, has to select the list of peers to play with. This service implements a trust algorithm where past experiences of the selecting peer are used to calculate the probability of trustworthiness of another peer; during an interaction, a peer observes the actions performed by other peers, compares them with committed actions and stores the result of this comparison in its database of past experiences. Next time it will interact with the peer, it will read this database and will search for records of similar scenarios (interaction contexts). The resulted values are used to compute the probability of trustworthiness.

5.3 Interaction lifecycle

In this section, we describe the main phases which the system goes through when a coordinated interaction between peers has to take place: subscription to an LCC-based interaction, bootstrap of the interaction, execution of the interaction.

5.3.1 Subscription to an interaction model

A peer willing to perform a task (e.g., verifying the flood state in some area, providing the water-level information service), searches for published IMs by sending a keyword-based query to the DDS (Figure 5.7, step 1) which in turn collects them by matching the query (eventually extended with synonyms to improve recall) and sends back the list to the peer (Figure 5.7, step 3);

Since interaction models and peers are possibly designed by different entities, the constraints and the peers' knowledge bases rarely correspond perfectly. The heterogeneity problem is addressed in two main phases [22]: (1) the DDS selects the interactions by matching their descriptions using a simple query expansion mechanism (Figure 5.7, step 2); (2) the peers compare the constraints in the received interaction models with their own capabilities [53](Figure 5.7, step 4).

The peers capabilities are provided by the plug-in OpenKnowledge components described in Section 5.2.1. OKC annotated methods are compared to the constraints in the interaction models, according to a mechanism which is identical to the one described in Section 5.2.1 to access the peer's local knowledge. The only difference is that, here, the mapping is performed between the annotated signatures of the constraints and that of the OKC methods. In this case, the adaptors created have a confidence level that represents the distance between the constraint and the best matching

method; the average of all the confidences of constraints gives a measure of how well the peer can execute an interaction, and it is used to select the most suitable one [22].

Once the peer has selected an interaction, it notifies its intent to play one of its roles to the discovery service, by subscribing to it (Figure 5.7, step 5).

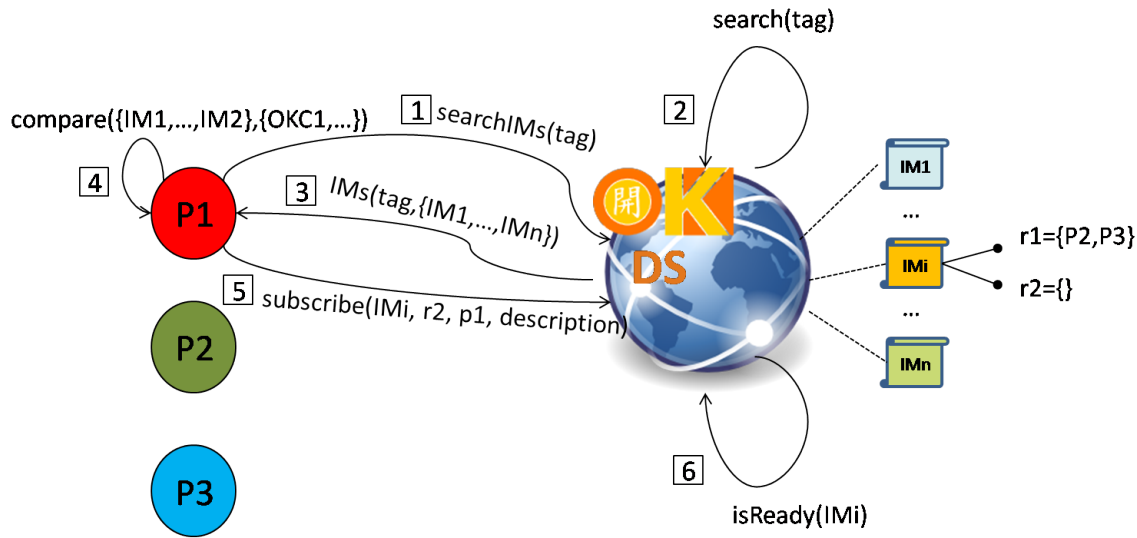


Figure 5.7: Interaction Subscription

Figure 5.8 [111] sketches the network status when the roles in interaction IM1 are all subscribed by at least one peer. The peers have installed their OKCs locally: some of them can be found online (e.g., OKC1, OKC2 and OKC3), others might be private to a peer (e.g., OKC4).

5.3.2 Interaction bootstrap

When all roles in an interaction model have at least one subscriber, the DDS selects randomly a coordinator peer from the peer-to-peer network (Figure 5.9, step 1), and sends it the information needed to start the team formation process for the selected IM (Figure 5.9, step 2). This process consists of the following steps:

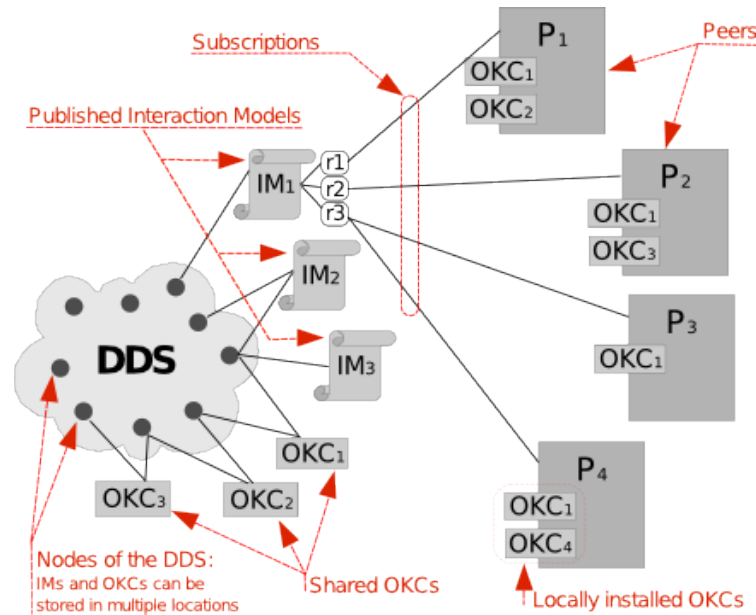


Figure 5.8: OK network at subscription time [111]

- The coordinator asks each peer in the list of subscribers to select the peers it wants to interact with (Figure 5.9, step 3);
- Adopting a specific selection strategy (e.g., randomly, trust score based, user based, etc.), each peer sends back its preferences (Figure 5.9, steps 4-5);
- The coordinator, when has received enough replies, creates a compatible team and sends the outcome to all subscribed peers (Figure 5.9, step 6).

It is important to notice that an interaction can begin at any moment but it can also never start if one role doesn't have subscriptions.

5.3.3 Interaction run

The coordinator peer runs the interaction model locally, by exchanging messages between local proxies of the peers. Every time a constraint has

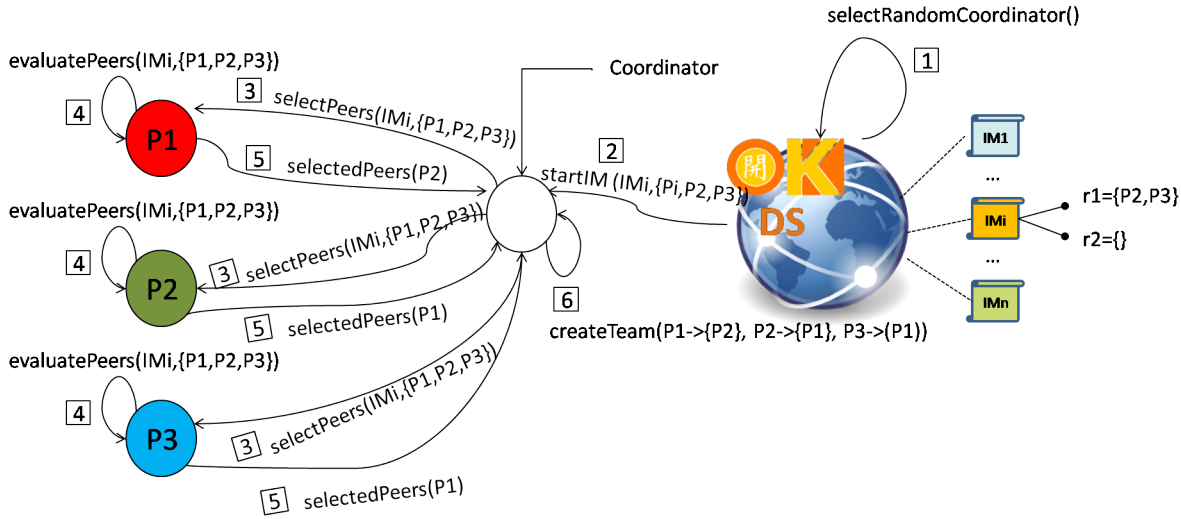


Figure 5.9: Interaction Bootstrap

to be solved, it sends a message with that constraint to the appropriate peer (Figure 5.10, step 1). The peer, therefore, solves the constraint by calling the corresponding method of the proper OKC (Figure 5.10, step 2) and then sends back to the coordinator a message with updated variables and a boolean value indicating whether the constraint has been satisfied or not (Figure 5.10, step 3). The interaction ends when either all messages of the interaction have been sent or some failures in a peer response occur. Finally, the coordinator sends an interaction feedback to all peers involved so that they can optionally send some observations that will be used by the trust component to compute the score needed during the selection phase of future interactions.

5.4 OpenKnowledge vs. other technologies

The OpenKnowledge platform aims at enacting interactions in a distributed P2P manner, the ultimate purpose being that of sharing knowledge between heterogeneous computational processes in an effective, free and reliable way. Under this view, the OpenKnowledge framework can be com-

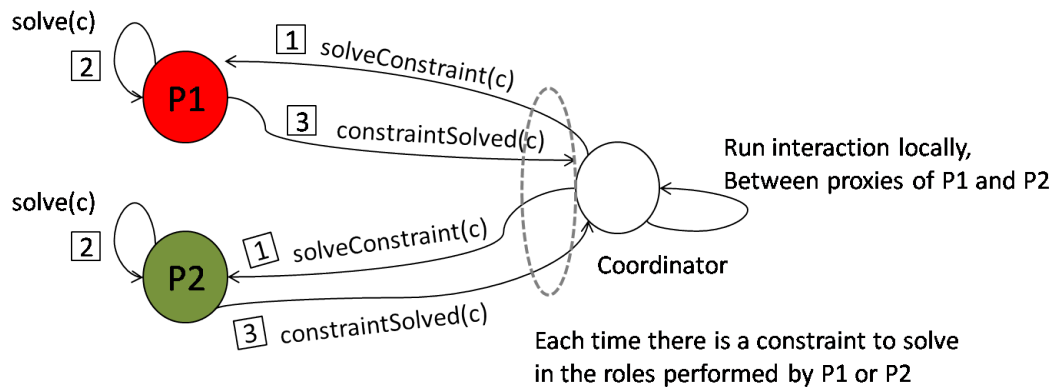


Figure 5.10: Interaction Run

pared and contrasted with the most prominent current technologies such as web services, grid services, p2p and the multi-agent systems[103]. In a way, the OK system integrates some of the features and functionalities which are present in the just mentioned technologies. In the next paragraphs, we briefly compare the OpenKnowledge approach with such technology mainstreams.

5.4.1 Web services

As already described in Chapter 2, one of the main concerns in the web service community is to build an open web environment where platform-independent software components are integrated and coordinated in a “loosely-coupled” way. In this respect, two main approaches exist: an orchestration approach and a choreography-based one.

In the first, single web services are composed into (business) processes which specify how they are going to interact with each other to accomplish the business task; a central service is responsible for the execution of the process. Semantic Web technology is often applied to ease web service discovery and automate composition, and, hence, to solve the semantic interoperability problem in a dynamic and distributed environment

[108]. Under this perspective, while (semantic) web services provide ways to automatically compose complex services out of simpler ones, in OpenKnowledge these are predefined workflows of services. In this sense, OpenKnowledge is more rigid; where it provides flexibility in respect to web services is in its ability to separate the advertising of a service (the interaction model) from the execution of a service [103], which is not possible with web services. Also, while in Openknowledge the storage of workflows is done in a distributed way, in the classical web service architecture this is achieved through central repositories. Moreover, while in semantic web services the matching between provider capabilities and requester's needs requires the establishment of a shared and agreed ontology, in OpenKnowledge the semantic interoperability problem is solved within the context of the interaction at hand.

In a choreography-based approach, the collaboration model is equally visible to all parties and is defined in terms of externally observable interactions existing between services. In this perspective, OpenKnowledge provides an implementation of the framework needed to allow the execution of choreographed interactions, thus contributing to the few existing platforms (see Chapter 2 , Section 2.2 for more details).

5.4.2 Grid services

Grid technologies aim at improving standard web services and were initially adopted to perform computing demanding task in a ditributed way [48]. As in OpenKnowledge, the coordination of activities is achieved by means of predefined workflows, but while in Grid systems these are centrally stored, in OpenKnowledge this is done using a p2p mechanism. Furthermore, like web services, grid services differ from OpenKnowledge by advertising a service functionality together with the identification of the service-provider; also, they focus on delivering functionalities such as long-term stability of

services, provenance, quality of service and resource monitoring, which are not considered by the OK framework [103].

5.4.3 P2P systems

OpenKnowledge is obviously tied with the principles of peer-to-peer systems: distributed storage, symmetry of roles among the peers, etc. However, two basic characteristics distinguish the OK system from traditional p2p architectures: (1) the focus on service-sharing rather than on data-sharing, and (2) the use of semantic matching approaches to service discovery which ascribe it to the category of semantic p2p systems [103].

5.4.4 Multi-Agent Systems (MAS)

Being OpenKnowledge a p2p framework, the analogy with multi-agent systems is straightforward: both frameworks aim at permitting non-trivial interactions between heterogeneous agents; and this is achieved in both systems by enabling contracts to which all agents conform. However, while in multi-agent systems peers are cognitive agents expected to be proactive and designed with complex architectures, in OpenKnowledge they are intended to be reactive components.

The above described analogies and differences between OK and other dominant technologies are schematized by Siebes et al. [103] in a table that we report here as Table 5.1.

Table 5.1: OpenKnowledge vs. Other Technologies

	Similarities	Differences	
		Web-Services	OpenKnowledge
Web-Services	service-oriented distributed automated search based on semantic descriptions	composition of atomic services fixed link to executing party centralised advertising equivalence matching	predefined workflows dynamic recruiting distributed approximate matching
Grid-Services	service-oriented fixed workflows distributed	Grid-Services provenance QoS resource monitoring centralised advertising fixed link to executing party	OpenKnowledge absent reputation mechanisms absent distributed dynamic recruiting
P2P Systems	distributed scalable symmetric roles of each peer	P2P Systems aimed at data-sharing independent of content	OpenKnowledge service sharing exploit semantics
Multi-Agent Systems	distributed symmetric roles of each peer	Multi-Agent Systems cognitive architecture central brokers pro-active behaviour	OpenKnowledge none scalable discovery reactive

Part III

Design and Simulation of LCC e-Response Scenarios

Chapter 6

Information Sharing in Crisis Response

In the first two parts of this thesis, we overviewed state-of-the-art research in coordination technologies and we presented OpenKnowledge, a framework addressing both the knowledge sharing and coordination issues that open and distributed information systems are required to tackle.

This chapter presents emergency response, as the application domain where the framework can be exploited and tested. The choice of emergency response as a target domain is driven by its potential to deal with a distributed-knowledge and dynamic environment. The nature of this domain requires a structured coordination in order to efficiently handle potential untidy and uncontrolled events. However, flexibility has to be taken into account when dealing with unexpected conditions (e.g., sudden road blockage, resource outage, landslide, etc.), which are prone to happen in emergency situations. Here, while the general vision of interaction protocols accounts for the “structured coordination” requirement of the problem, the adoption of a p2p framework for enacting interaction models can address the need for flexibility and dynamicity.

The chapter is organized as follows: Section 6.1 first introduces the main requirements emergency response systems need to support, as well

as the most common approaches used to design them; it then frames our research work in this context. Section 6.2 directs the focus on a specific flood disaster scenario, which is described by the main entities involved and their coordination.

6.1 The general context

As sketched in the introduction, response to disaster scenarios involves activities developed and implemented through the essential analysis of information and the coordination of different actors (both institutional, like emergency personnel, army, volunteers, and so forth, as well as common people involved in the crisis).

The existence of numerous, geographically sparsed and different actors, policies, processes, data standards and systems, results in coordination problems regarding information gathering, data analysis and resources management, all critical elements of emergency response management. In particular, information gathering is essential to increase the effectiveness of the response; in fact, experience taught us how many response failures came about because information was never collected or was unknown, thus obliging people to take decisions blindly [24].

In order to make information sharing effective, a shared understanding among the various stakeholders is needed, despite each community maintains its own view of the domain, i.e., gives meaning and classifies terms in different ways [41]. For these reasons, it is unrealistic to “*monolithically agree on a single representation of all the knowledge that will be involved*” [109].

Moreover, it is very difficult to know a priori which actors will assume which roles and what actions they will plan in the immediate future [88, 114]. This often leads to ad hoc teams of emergency officers from different

organizations working together on sometimes unpredictable and novel tasks [61]; in addition, during emergency response¹ activities, it is often the case where transitions of functions between roles and organizations occur [3].

In view of what has been just mentioned, some of the requirements that e-response systems need to address can be summarized as follows:

- Decentralized control towards response strategy;
- Effective information gathering;
- Semantic agreement among different stakeholders;
- “On-the-fly” formation of e-response teams;
- Coordination of activities;
- Scalability;
- Flexible but structured interactions;
- Requests for assistance and matching to available capabilities, role change.

It is straightforward to conceive the OpenKnowledge approach as a viable framework for the satisfaction of the above requirements; hence, the OpenKnowledge system as a possible technology for their implementation.

Approaches to design crisis response systems mainly include crisis drills and simulation tools [64]. While drills are expensive and difficult to mimic large-scale disasters, it is recognized [60, 15] that simulation tools can be valuable for investigating innovative solutions, such as new collaborative information systems, new cooperation configurations and communication devices.

¹In the rest of this thesis, we will refer to *emergency response* as *e-response*.

The National Institute of Standards and Technology² (NIST) has proposed an Integrated Emergency Response Framework (iERF) where three axis are defined in order to classify such tools [60], that are: the application for which the simulation tools are developed, the entities involved, the kind of disaster. These factors can affect the development of the simulation environment. Figure 6.1 depicts the iERF framework.

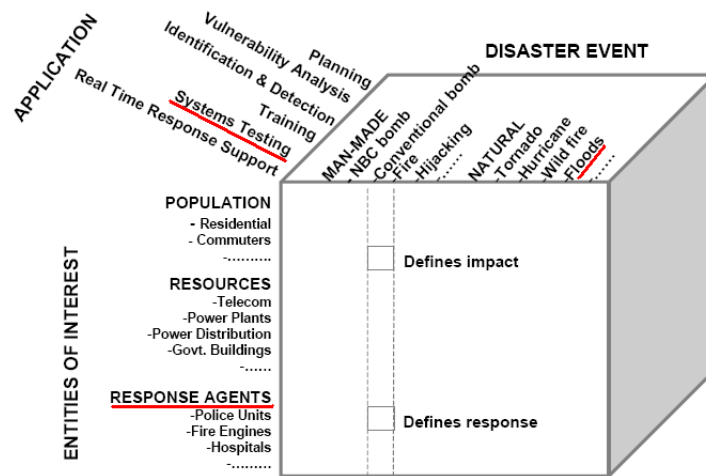


Figure 6.1: Integrated Emergency Response Framework (iERF) proposed by NIST

As can be observed from the figure, some elements are underlined in red: these are the ones considered in our work, which includes a software simulation where response agents need to gather information during a flood event; and that is used to exploit and test the coordination mechanism provided by the OK system.

The developed simulation environment will be described in Chapter 8; the next section describes the specific scenario where OpenKnowledge has been tested and proven to support coordinated e-response activities.

²<http://www.nist.gov/index.html>

6.2 Flooding in Trentino: the e-response case study

We exploited and tested the OpenKnowledge framework in the case study of a flood disaster in Trento (Italy). The work started from a preliminary analysis on this kind of disaster. The available analysis resulted from documents related to the current flood emergency plan in the Trentino region and from interviews with experts.

We individuated emergency peers (e.g., firemen, police, medical, bus/ambulance agents, etc.), the main organization involved (e.g., Emergency Coordination Center, Fire Agency, Civil Protection Unit, Provincial Health Agency, etc.), a hierarchy between the actors (e.g., emergency chief, subordinate peers, etc.), service peers (e.g., water level sensors, route services, weather forecast services, GIS services, etc.) and a number of possible scenarios, that is, possible interactions among the agents and their assigned tasks.

The peers can be distinguished into two main categories: *service peers* and *emergency peers*. While the former are basically peers providing services under request, the latter are peers often acting on behalf of emergency human agents that are in charge of realizing the emergency plan. A comprehensive description of all peers and tasks can be found in OpenKnowledge deliverables [115] and [116]. Figure 6.2 recalls the richness of all scenarios and interactions possibly involved. The areas circled in red, concern the scenarios actually modeled in terms of LCC interactions. In what follows, we illustrate only such scenarios. The upper part of the figure represents the *pre-alarm* phase of the emergency plan foreseen by the Autonomous Province of Trento. The lower part relates to the *evacuation* phase.

In the prealarm phase, the involved peers are mainly service peers which are, as has been previously described, peers providing the information upon which the decision on whether to enact the emergency plan or not is made.

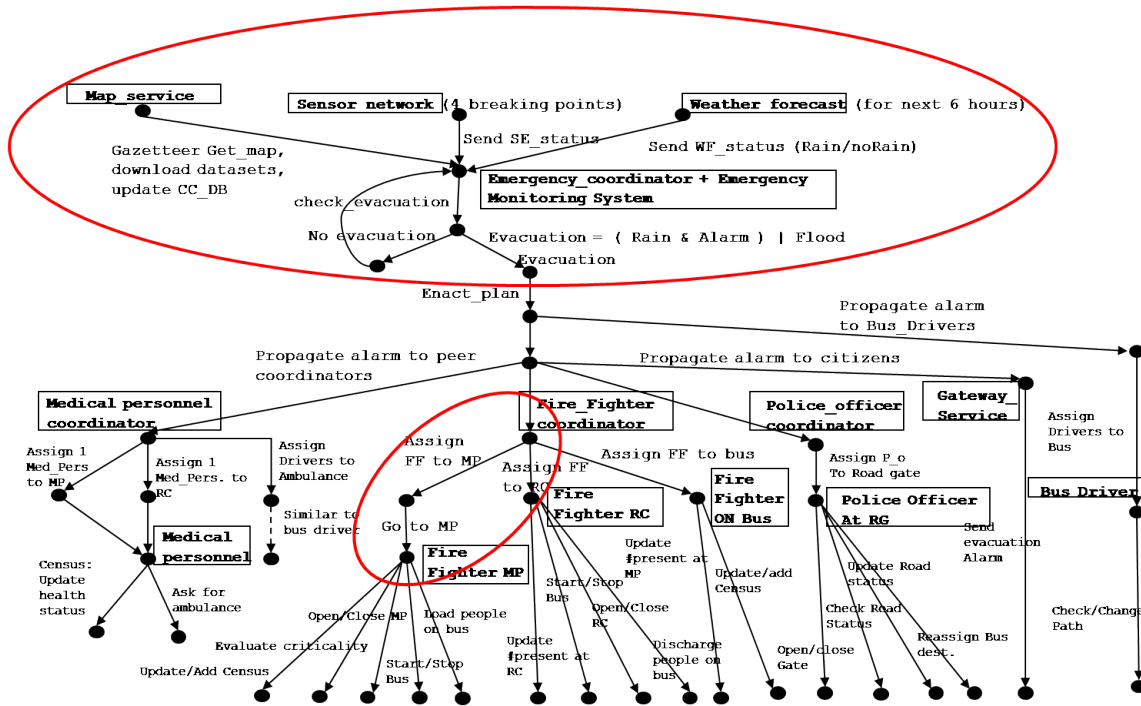


Figure 6.2: The overall e-Response use case

The pre-alarm phase, hence, mainly involves service peers which provide information useful for decision making. It eventually results in the evacuation phase, which regards all the activities needed to move people to safe places. In such evacuation phase, the key peers are emergency peers, that is, all the peers in charge of helping in the evacuation of citizen: emergency coordinators, firemen, government agencies (e.g., civilian protection department), real-time water level data reporters (e.g., people, sensors). The emergency peers are supported by service peers such as route services, sensors scattered across the emergency area, etc.

Figure 6.3 gives a schematic view of the two phases involved in our case study. It shows the involved actors (denoted by round circles), their interactions and the kind of information exchanged. The smooth rectangle denotes the simulator, that is, the virtual environment where all the peers

act; obviously, it doesn't correspond to any entity in the reality, therefore, we don't describe it in this context. However, the simulator is essential for the simulation-based testbed and will be illustrated in detail in Chapter 8 (Section 8.2.2).

The figure also shows two different evacuation sub-scenarios: in both of them, an emergency subordinate (*ES*) needs to get information on route's practicability but while in one case (area above the red line), it gets the route blockage status by asking the Civil Protection (*CP*), in the other case (area below the red line), it interacts directly with reporters (*r1,r2,r5*) physically present at the locations of interest. These two ways of gathering information are referred to as *centralized* and *decentralized* strategies.

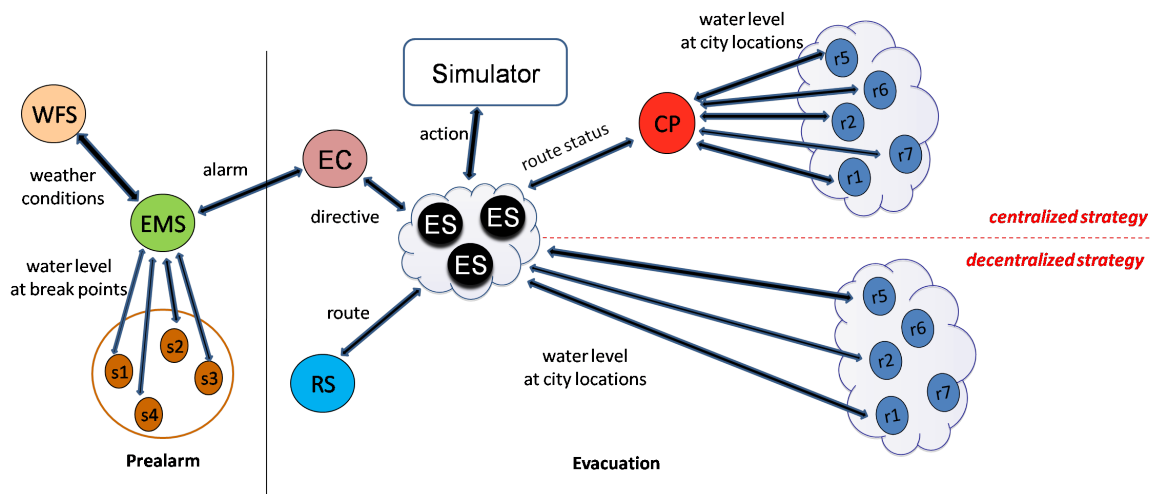


Figure 6.3: The implemented e-Response scenarios

6.2.1 Prealarm scenario

In the pre-alarm phase, the water level of critical points along the river is constantly monitored by an emergency monitoring system (EMS). Such system also checks weather information in order to enrich the data needed to predict the evolution of a potential flooding. When a critical situation

is registered, the emergency chief is notified, then to be able to take the proper actions.

Peer types

The peer types involved in the pre-alarm scenario are the following:

- *Emergency Monitoring System (EMS)*: such system represents the server station where all the information which are critical to the emergency are collected. In particular, the system:
 - collects weather forecast information;
 - collects water level information from sensors located along the Adige river;
 - analyses the previous information;
 - when needed, sends a proper alarm message to the emergency coordinator;
- *Water Level Sensor(S)*: represents a water level sensor placed in one of the four strategic points along the Adige River; provides water level information registered at the location where it is placed;
- *Weather Forecast Provider (WFP)*: provides weather conditions (i.e., temperature, rain probability, wind strength) given a specific location.
- *Emergency Chief (EC)*: the top-level authority which is notified by the EMS and is in charge of making decisions.

As can be noticed, the majority of the above peers are mainly what we denoted as “service” peers.

Prealarm interactions

The scenarios pertaining the prealarm phase and which are modeled in terms of LCC interaction models are two. A short description for each of them follows:

1. *Monitoring of water level sensors*: a central monitoring system (EMS) continuously requests information on the level of water registered at critical positions along the river. Such information, together with the one about the precipitation rate in the next days, is crucial to enact the evacuation plan;
2. *Weather info collection*: the emergency coordinator request periodically a weather forecast (i.e., rain and temperature) in order to make previsions and therefore decisions on the actions to take.

6.2.2 Evacuation scenario

As anticipated before, the evacuation plan consists of peers (e.g., firemen, buses, citizen) moving to safe locations. In order to move, such peers need to perform activities such as choosing a path to follow (usually by asking a route service), checking if the path is practicable (usually by interacting with the Civil Protection or with available reporters distributed in the area), proceeding along the path. The Civil Protection can deliver information on the blockage state of some given path to a requester. It is able to do that since it is continuously gathering information from reporters scattered around the emergency area. Such reporters inform on the water level registered at their locations.

Peer types

The peer types involved in this scenarios are the following:

- *Emergency Chief (EC)*: such peer is responsible for the coordination of all the emergency activities, from the propagation of the alarm to its subordinates, to resources allocation. Specifically, it:
 - receives different levels of emergency alarm messages from the EMS;
 - collects GIS information;
 - collects specific weather information (e.g., temperature, rain probability, wind strength, etc.);
 - sends directives to its subordinates (e.g., move to a specific point, close a meeting point).
- *Emergency Subordinate (ES)*: is a peer (e.g., an emergency subordinate as a fireman, a bus, a citizen) in need to move to a specific location;
- *Route Service (RS)*: is able to provide a route between two points that takes into account roads that might be flooded, blocked or otherwise inaccessible. It does this by having a map of the area and through running separate interactions with other peers to keep track of the current state of the roads;
- *Civil Protection (CP)*: is responsible for giving information on the blockage state of a given path;
- *Reporter (R)*: is responsible for giving information on the water level registered at its location. It could be either a citizen or a sensor device permanently placed at a given location. In our simulations, we consider reporters as fixed sensor devices.

Evacuation interactions

The main interactions involved in the evacuation scenario are shortly described below:

1. *Evacuation*: describes how an evacuation plan evolves. An emergency coordinator alerts members to go to a specific destination. Each member finds a path to reach the destination, checks its status and eventually moves along the path;
2. *Find a route*: describes the interaction needed to retrieve a path from a route service;
3. *Check path status with CP*: describes the interactions with the Civil Protection needed to know the blockage state of a path;
4. *Gather real-time data from reporters*: a peer requests information about the water level from a group of reporters.

Baseline scenario

As mentioned before, part of the evacuation scenario consists of checking whether a given route is practicable or not. However, we also considered the case in which the peer chooses to directly move along the route without getting any information on its blockage conditions. This scenario, in which no strategy is adopted in order to gather such useful information, constitutes the *baseline* scenario: a route is taken without checking a priori its conditions.

Information gathering scenarios

Although the baseline scenario is realistic, one of the aim of this work is to test the capability of the OK framework to support the two information

gathering strategies mentioned earlier, i.e., the strategies relating to how an emergency subordinate gathers information on the route's practicability:

- **Centralized strategy:** a moving peer (e.g., an emergency subordinate) obtains information on the blockage conditions of a given route after consultation with the Civil Protection *CP* (see Figure 6.3 - area above the red line);
- **Decentralized strategy:** a moving peer obtains information on the blockage conditions of a given route by gathering water level information from a selected group of reporters (see Figure 6.3 - area below the red line).

What we want to underline here is that the adoption of an information-gathering strategy benefits the peer in committing to its duty. Figure 6.4 shows the behaviour adopted by the moving peer while moving along a path. Every time it reaches a location, the peer gets information on the blockage state of the route ahead. Notice that, when the path is blocked because of an excessive level of water in a location, the moving peer is aware of that in advance and is, thus, able to find an alternative path before approaching the blocked location.

More details on how the moving peer reasons about the information thus gathered is given in Section 8.2.1, where the peer-network component of the e-Response simulation system is described.

6.3 Conclusion

Emergency response is not inherently peer-to-peer: we could expect that the key players would have strategies worked out well in advance and would have established the infrastructure and vocabulary for communicating with other key players. However, the chaotic nature of an emergency means that

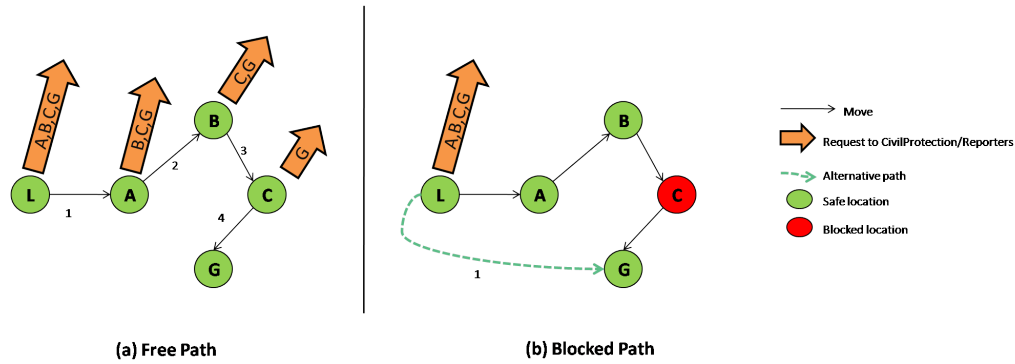


Figure 6.4: Evacuation phase: moving peer behaviour

many players who will not have been able to coordinate in advance, or who were not expected to participate, may become involved. Furthermore, services who were part of an emergency response may be unexpectedly unavailable or may be swamped by requests, and in such a situation, it is crucial that the emergency response can carry on regardless. Additionally, services may develop and change and it is unrealistic to expect these changes would always be known and accounted for in advance.

For example, in the described flood disaster, the routing service may have altered over time and have failed to keep the service up to date with the changes; in other situations, it could be that the civil protection lost connection or was swamped with requests.

In the above scenarios, an infrastructure like the one provided by the OK system would allow a fire team to use the discovery service to locate a new routing service, or to switch to a distributed information gathering strategy in case the civil protection becomes unavailable. Still, once teams of peers are formed, the use of simple and predefined coordination tasks would facilitate their interactions.

Chapter 7

LCC Protocol Design

The scenarios described in the previous chapter were translated in terms of LCC interaction protocols. The full LCC specifications related to the pre-alarm scenario can be found in the OpenKnowledge project deliverable [113], while the ones used for the evacuation scenario can be consulted in Appendices A-B.

This chapter focuses more on the process that lead from the analysis of the scenarios to the design of LCC protocols and, consequently, to the requirements suggested for the development of the OK system. The chapter is organized in three sections: Section 7.1 introduces the initial design of the LCC protocols, carried out when the OK system was not fully in place; it describes the design of a relatively complex evacuation scenario, illustrates the inherent limitations of the LCC interaction models and identifies the requirements for the design of complex and realistic coordination tasks. Section 7.2 illustrates the second phase of the protocol design, carried out after these requirements were addressed by the OK infrastructure. Finally, Section 7.3 concludes the chapter.

7.1 Initial design

Prior to the specification of emergency response LCC interaction models, we designed UML diagrams to clarify the activities of the participating entities and their relationships. Figure 7.1 depicts an activity diagram that graphically sketches a baseline scenario.

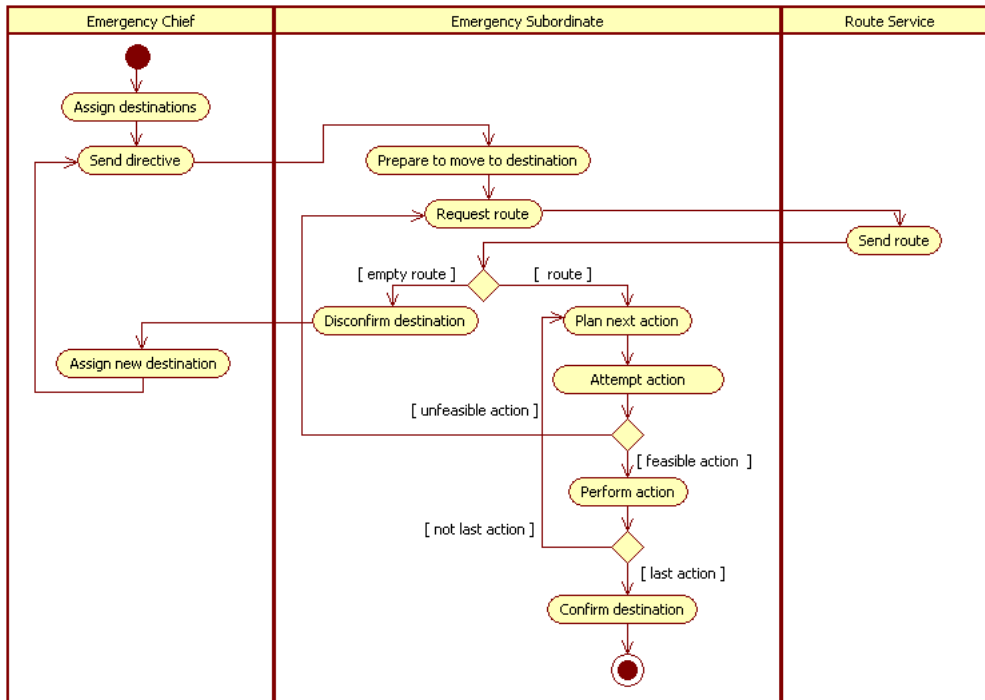


Figure 7.1: UML activity diagram: baseline scenario

This activity diagram clarifies the activities of the single e-response peers but hides the sequential order of their interactions. To explicit the details of the peer interactions, we designed UML sequence diagrams like the one given in Figure 7.2.

The sequence diagram shows the sequence of messages exchanged between the e-response peers, and also shows the interaction between an emergency subordinate and the environment, that takes place when an action is performed.

7.1. INITIAL DESIGN

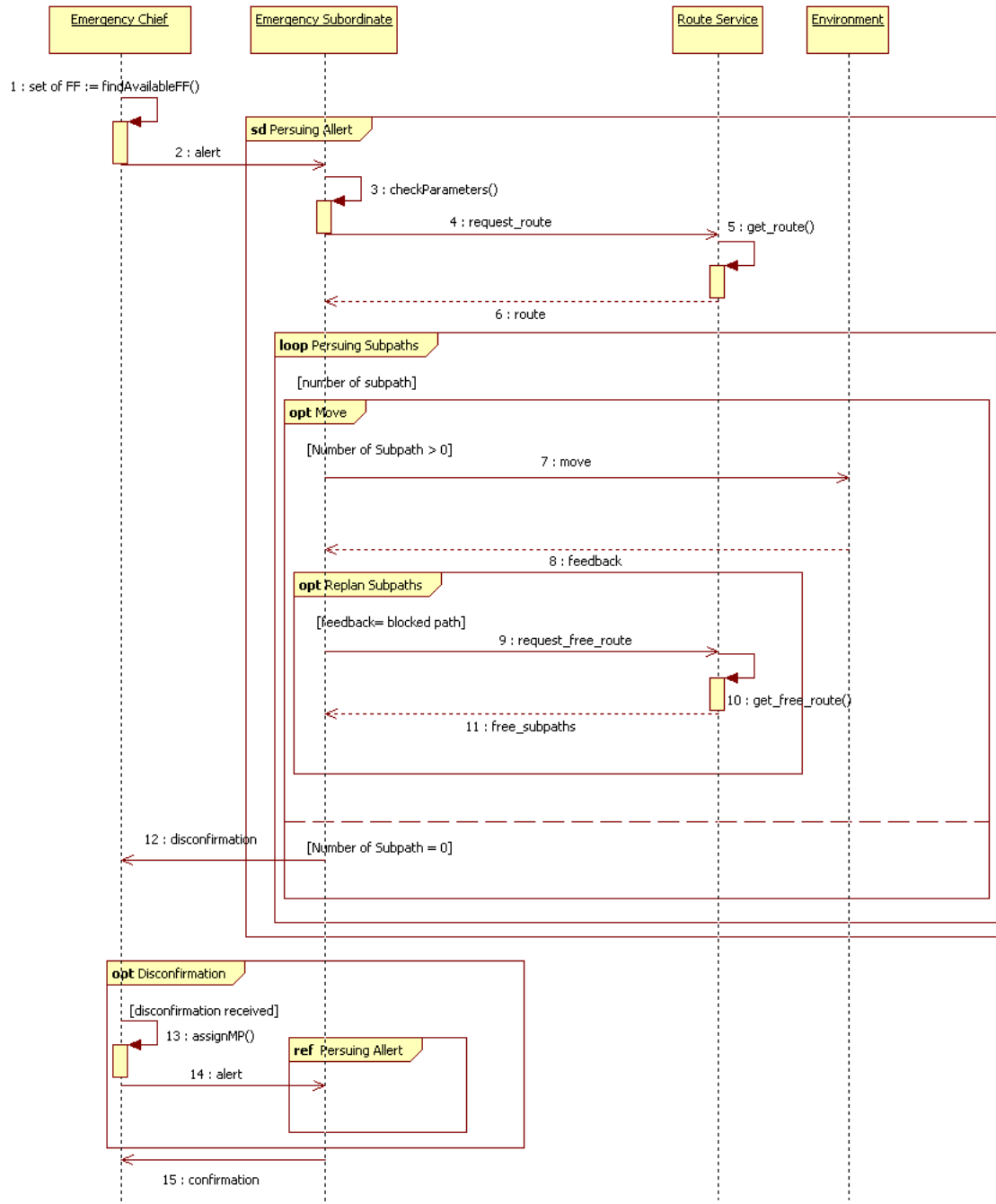


Figure 7.2: UML sequence diagram: baseline scenario

Although we are modeling a relatively simple scenario, the figure gives a flavour of how complex the interaction patterns can be. Starting from

```

a(emergency_subordinate, ES) ::
  alert(MP) ← (emergency_chief, EC)
  then a(route_finder(MP, EC), ES)

```

Figure 7.3: LCC fragment: *ES* initial role

```

a(route_finder(Dest, EC), ES) ::
  null ← getPos(Pos) and not(Pos = Dest) and getRouteService(RS)
  then request_route(Pos, Dest) ⇒ (route_service, RS)
  then (
    route(From, To, Path, SubPaths) ← (route_service, RS) then
    null ← store(Path, SubPaths) then
    a(action_performer(move(From, To, Path), ES)
  )

```

Figure 7.4: LCC fragment: *ES* “route finder” role

these diagrams we engineered LCC protocols specifying the roles of all participating entities, the environment (e.g., the simulator) included. This resulted in one complex interaction model that can be found described in the OpenKnowledge project deliverable [71].

Below, we briefly outline some of the roles taken on by the emergency subordinate *ES*. Figure 7.3 illustrates the initial role, when an alert message stating to go to the destination *MP* is received from the emergency chief *EC*. Once the message is received, the emergency subordinate, *ES*, assumes the role of *route_finder*. In such role (see Figure 7.4), a route between the current position and the final destination is requested to the route service *RS*. When the path is received from the route service, the *ES* stores it together with the sub-paths and then attempts to move by entering the *action_performer* role. This last role, whose definition is not reported here, represents the interactions of the *ES* peer with the surrounding environment, and includes the process of replanning the actions (e.g., request an alternative path); such process is enacted depending on the circumstances, for example, when some locations are blocked by the flood.

As it can be noticed from the depicted LCC fragments, different tasks

are mixed together in one single interaction model: the communication with the route service, the sensing from the environment, the replanning of actions. Furthermore, a peer must know in advance the services with which to interact: the constraint *getRouteService(RS)* consults the route services known by the *ES* peer.

It is clear that, with this kind of protocol design, the modeling of more complex scenarios becomes unfeasible, and not even useful. In fact, for example, to enact a distributed information gathering strategy, it is necessary for a peer to interact with only those reportees located along the path that has to be followed; obviously, such reporters cannot be known in advance. Also, as already mentioned, route services might become unavailable, and so forth.

Besides this, in a real world scenario, an interaction model designed in such a way would not be of great use. In fact, one would expect to enact simple and well-targeted tasks; for example, the request for a route could be specified in one self-contained task involving only two participating entities, the service requester and the service provider. The possibility to compose protocols in a modular way, thus becomes a necessity, in two aspects: the first regards the protocol design and the ability to specify complex models of interactions in a clean way; the second aspect, which is linked with the first one, regards the exploitation of such models in real world contexts, as we just mentioned.

The lack of modularity, in the protocols initially designed, is mainly due to the fact that the composition is not an intrinsic feature of the LCC language; also, the underlying enactment infrastructure (i.e., the OK system) could not offer the possibility to enact separated coordination tasks within an ongoing interaction.

We therefore identified the following requirements:

- the possibility to enact separated interactions, thus to be able to design more composite interactions;
- the possibility to adopt peer strategies for the selection of specific peers;

It is important to notice here that during this first stage of protocol design, the OK system was still under development. The above requirements were thus suggested to the team involved in the system implementation. As a result, the *peer access mechanism* already illustrated in Section 5.2.1 was devised to enact “nested” interactions.

7.2 Final design

In this section, we present the design of a more complex evacuation scenario, the one including the information gathering strategies. The main interactions included in this scenario are described in Section 6.2.2 and represented by the UML diagram of Figure 7.5. In this diagram, the activities needed to check the path conditions are grouped together, for clarity.

The designed LCC protocol, named “Evacuation”¹, is described here in its main parts. As in the baseline scenario, the emergency subordinate *ES* receives an alert message from the chief and resolves the constraints needed to set the goal to be achieved, and to get the current position. The activities of *ES* thus evolve through three key LCC roles: the *goal achiever* role which abstractly models the activity of searching for a path and moving towards the goal; the *free_path_finder* role which defines the operations needed to find a free path; the *goal_mover* role which models the actions needed to move towards the goal destination. Figures 7.6-7.7

¹This IM is described in full details in Appendix A.1

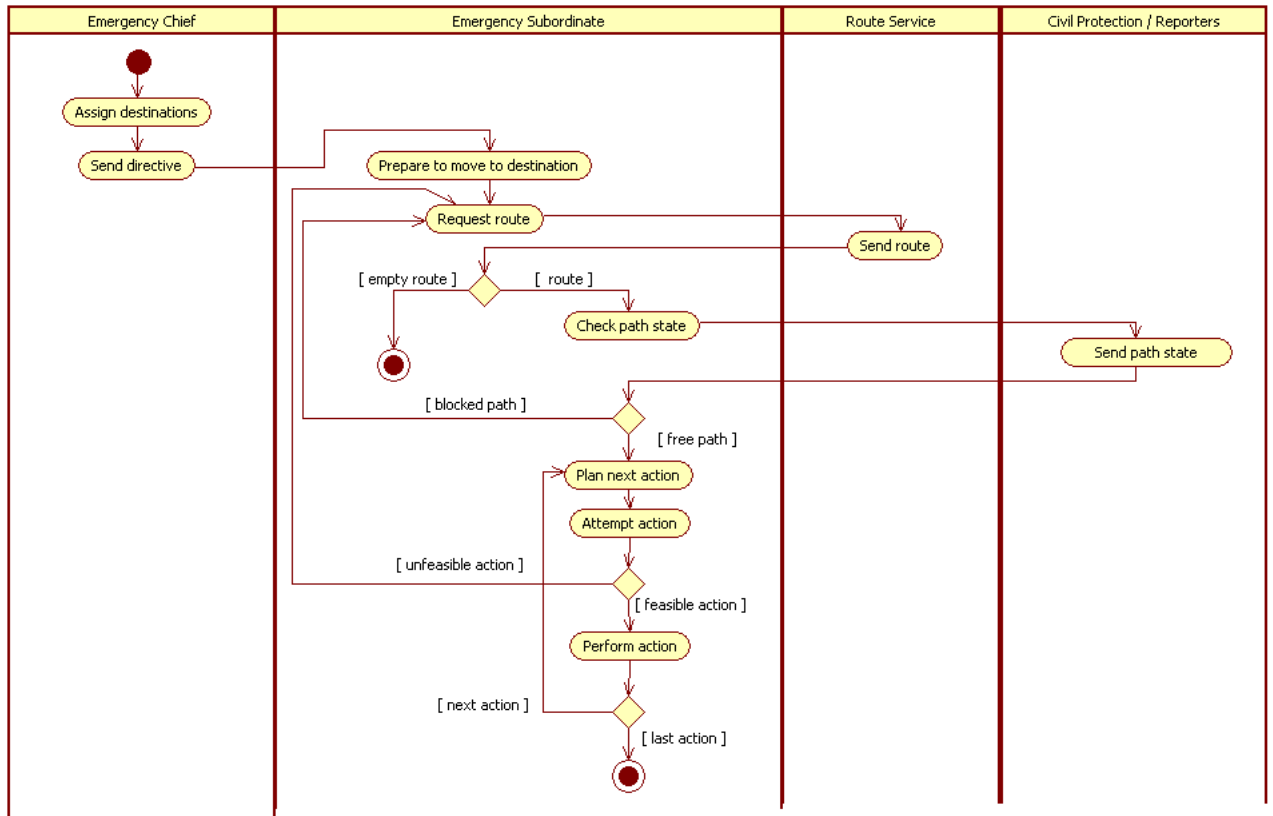


Figure 7.5: UML activity diagram: evacuation scenario

show LCC code snippets for two of the key roles. The constraints specified in bold are the ones enacting separate interactions.

For example, the constraint $find_path(From, To, Path)$ of Figure 7.7 enacts the “Find-Route”² interaction, i.e., the coordination task needed to request a path from a route service. The constraint $request_path_state(Path, PathState)$ shown in the same figure, eventually enacts either the “Check-Route-State” interaction or the “Querier-Reporter” interaction, depending on the adopted information gathering strategy. Finally, since this is a simulated model, the constraint try_move_action of the *goal_mover* role³, enacts the interaction needed to sense the information from the environment, hence, to

²More details on the “Find-Route” LCC protocol can be found in Appendix A.2.

³This role is fully explained in Appendix A.1.

```

a(emergency_subordinate,FF)::

alert(G) <= a(emergency_chief,FFC) then
  null <- set_goal(G) and get_current_position(CurrPos) then
    a(goal_achiever(CurrPos,G),FF)

a(goal_achiever(From,To),GA)::

(
  //moving peer already at destination
  null <- equal(To,From) and setGoalAchieved(To)

  or

  ( //try to find a free path
    a(free_path_finder(From,To,FreePath), GA) then

      //no free paths between From and To
      null <- FreePath=[] and setGoalUnreachable(To)

      or

      //move towards the goal destination along the free path found
      a(goal_mover(From,To,FreePath),GA)
    )
  )
)

```

Figure 7.6: LCC fragment for the “goal-achiever” role

```

a(free_path_finder(From,To,FreePath), FRF) ::

null <- find_path(From,To,Path) then
(
  //no paths are found
  null <- Path=[] and makeEmptyList(FreePath)

  or

  (
    //check if the path is free
    null <- request_path_state(Path,PathState) and
    path_free(PathState) then
      null <- assign(Path,FreePath)
    )
  )

  or

  //search for an alternative path which is free
  a(free_path_finder(From,To,FreePath), FRF)
)

```

Figure 7.7: LCC fragment for the “free-path-finder” role

interact with the simulator.

It appears that the described design, makes feasible the modeling of complex interactions such as the ones under study; such design, in fact, permits to write simple, modular and reusable interactions. The enactment of LCC protocols designed in the way just presented, is made possible by

an enhanced release of the OK system.

7.3 Conclusion

This chapter has shown how the initial design of LCC protocols to model relatively complex scenarios has affected the development of the OK infrastructure. The OK system has in fact been developed contemporarily with the emergency response LCC interaction models; some of its features were suggested by the design requirements and the case study domain. Given the complexity of emergency response coordinated activities, it was necessary to provide a way to combine together modular and reusable interaction models.

Furthermore, the present chapter has illustrated a possible way to specify multiple interleaved interactions, in more complex emergency response scenarios such as the ones where information-gathering strategies are preferred over more “blind strategies”.

Chapter 8

The Simulation Environment

This chapter describes the simulation environment built to validate the interaction models, to use and test the current release of the OK infrastructure and to analyse centralized vs. distributed information gathering in the flood disaster case study.

The current simulation environment is based on a first version of the system, which was running on a Prolog based LCC engine [69]. The system we initially developed is basic, in the sense that it simply provides means to execute LCC interaction models. However, to run an interaction model, the peer has to know which interaction model it wants to execute and with which peers it needs to interact. The current simulation system extends the initial prototype both in a complete integration with the OK system and in the inclusion of a realistic flood-simulator. In particular, the following additional features can be found in this current version of our simulator:

1. Full integration with the OK kernel: the previous prolog simulation was ported and further extended into Java so to make full use of the OK Kernel , hence, the LCC Interpreter, the Discovery Service, and the Ontology Matching modules;
2. Dynamic evolution of flood: while in the previous simulation the blockage state of a node was fixed a priori, a realistic flood simulation

is embedded in the current system;

3. Modular/simple IMs: in the developed simulation environment, about ten single and independent interaction models are used, instead of a unique and relatively complex one (as in the previous simulation);
4. Increased peer types: the simulation system extends the previous one in the number of peer types involved in the emergency scenario. New peer types such as reporters and Civil Protection are considered;
5. Different information gathering strategies: the current simulation system extends the previous one in the scenarios involved; while, previously, the moving peer was meant to go directly to the destination assigned, in the present simulation the peer can adopt either a centralized or a distributed information-gathering strategy to request information on a route's state.

The chapter is organized as follows: Section 8.1 proposes a schematized overview of both the LCC protocols designed to model e-response tasks and those built to implement the simulator. Section 8.2 describes in more detail the architecture of the simulator and the interaction mechanisms which take place. Section 8.3 points out how OK components (i.e., LCC specifications and OKCs) are reused. Finally, Section 8.4 concludes the chapter.

8.1 LCC protocols: an overview

Figure 8.1 shows a complete list of all interaction models implemented for both the pre-alarm and evacuation phases. It also shows the type of peers involved and the separated interactions eventually enacted by a constraint in a given IM. The last column of the table indicates for which kind of

information gathering strategy a given IM is used. As can be noticed, all interaction models are used in both centralized (*C*) and decentralized (*D*) scenarios, except one: the “Check Route State” IM, which is only used in the context of the centralized strategy to interact with the Civil Protection *CP*. The figure also shows that the interaction models are modular and reusable in different contexts.

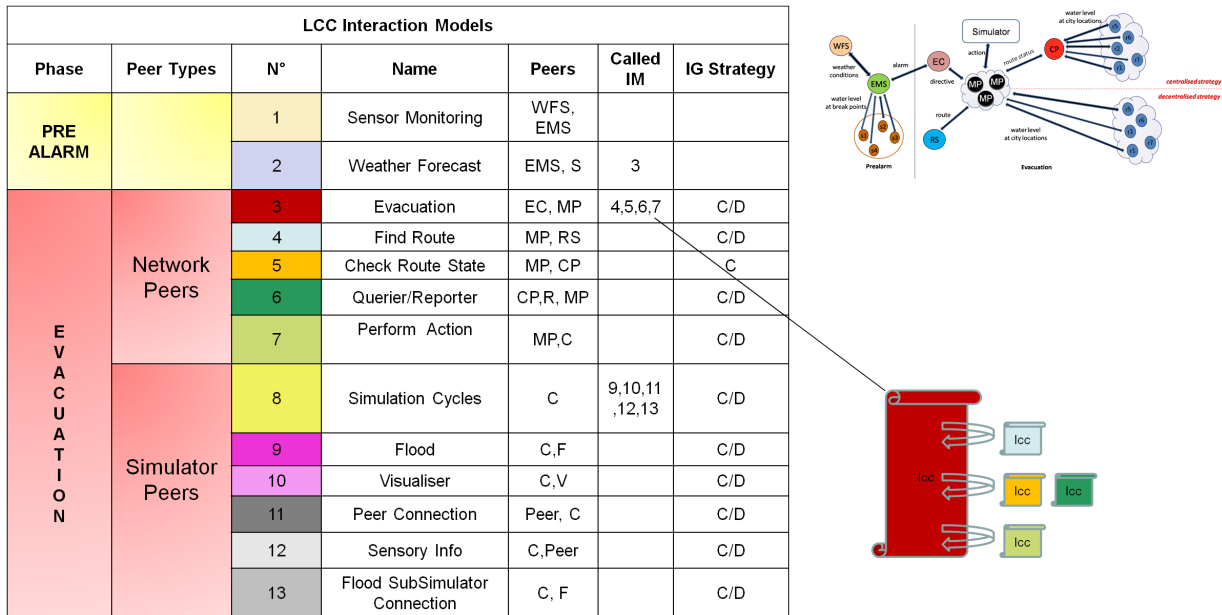


Figure 8.1: e-Response Interaction Models

8.2 Simulation architecture

Our simulation environment is composed of two main components: the peer network and the e-Response simulator. Figure 8.2 sketches its overall architecture. All peers are equipped with their own OpenKnowledge plug-in component(s); each black arrow represents a different interaction model, which also represents the flow of information between peers; the grey arrows indicate interactions among network peers only. In the next two subsections, we illustrate the peer network, and the core components

of the simulator, respectively.

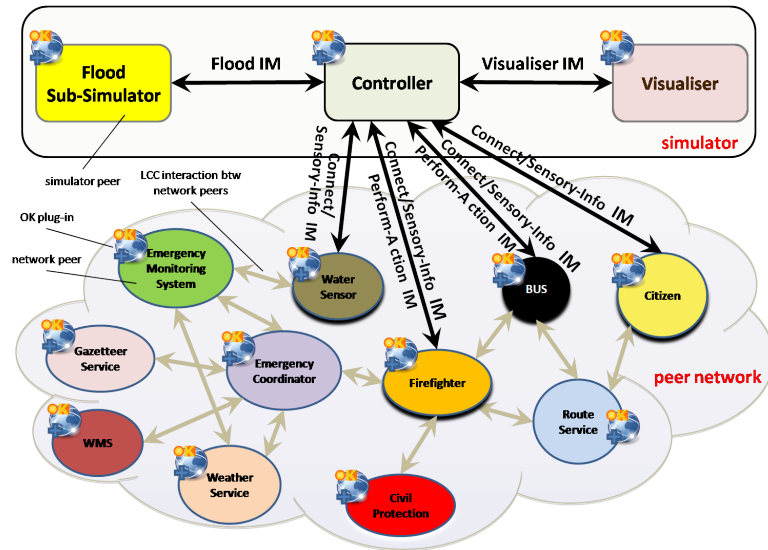


Figure 8.2: The e-Response system's architecture

8.2.1 The peer network

The peer network represents the group of agents involved in a simulated coordination task. An agent in the peer network can interact with other agents, perform actions (e.g., moving along a road) and gather information (e.g., sense the water level in its vicinity).

In order to perform an action or receive sensory information near its location, a peer must connect to the simulator by enacting the “Connect” interaction model. Once added to the simulation, the connected peer periodically receives sensory information from the simulator via the “Sensory-Info” interaction model; finally, to perform an action, a connected peer enacts the “Perform-Action” interaction model which models the action coordination with the simulator. The connected network peers are called *physical peers* (shaded ellipses in Figure 8.2).

Not all peers must connect to the simulator: *non-physical peers*, such as a route service that provides existing routes, do not need to communicate

with the controller but only with other peers in the peer network. In the real world such peers would not actually be in the disaster area and could not affect it directly, but could provide services to peers that are there. Non-physical peers are represented as non-shaded ellipses in Figure 8.2.

The interactions between the network peers which regard the evacuation phase, have already been described in the previous chapter. Figure 8.3 shows how these interactions are interleaved with the ones needed to connect with the simulator.

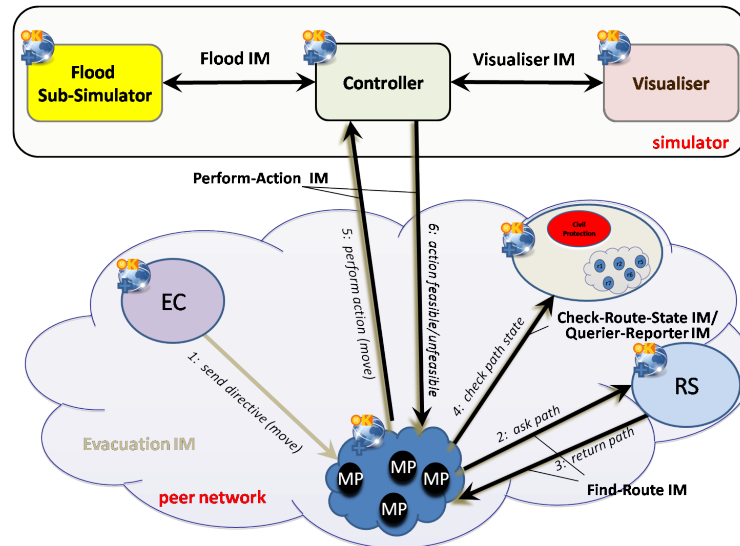


Figure 8.3: Evacuation phase: network peer’s interactions

The simulated coordination task evolves through the following ordered sequence of steps:

1. *Send directive*: the emergency chief EC sends the directive to move to a given destination to a moving peer *MP*;
2. *Ask route*: the *MP* asks a path to the route service *RS*;
3. *Return route*: the *MP* receives a path from *RS*;
4. *Check route state*: the *MP* checks the route state with either the Civil

Protection CP (centralized scenario) or the reporters r (decentralized scenario);

5. *Perform action*: the MP checks the feasibility of the (move) action;
6. *Return action feasibility*: the MP comes to know whether the action has been performed or stopped.

Eventually, steps 2 through 6 are repeated until the final destination is reached. The above sequence of actions is specified in terms of the “Evacuation” interaction model. This IM is the one capturing the whole evacuation scenario and has already been described in Chapter 7, Section 7.2. In Figure 8.3, grey arrows refer to activities entirely performed within the “Evacuation” IM while the black arrows shaded in grey indicate that the associated steps are executed by solving LCC constraints (in the main IM) which, in their core part, enact separate LCC interactions. As underlined previously, this is a key functionality of the OK system, which permits to write simple, modular and reusable LCC specifications.

Steps 2-3 pertain to the “Find-Route” IM¹, which is enacted by the constraint $find_path(From, To, Path)$ of Figure 8.4, that we show again for the reader’s convenience.

Step 4 is performed within the constraint $request_path_state(Path, PathState)$ shown in the same figure. Such constraint eventually enacts either the “Check-Route-State” or the “Querier-Reporter” IM, this depending on the information gathering strategy adopted. Finally, steps 5-6 regard the interactions specified in the “Perform-Action” IM², which is enacted within the constraint try_move_action of the $goal_mover$ role³;

We describe in detail the activity of checking the path state, since it represents the core part of our simulation. The constraint $request_path_state(Path, PathState)$

¹The LCC specification of this IM is to be found in Appendix A.2.

²The LCC specification of this IM is to be found in Appendix B.7.

³This role is fully explained in Appendix A.1.

```
a (free_path_finder (From, To, FreePath), FRF) ::
null <- find_path (From, To, Path) then
(
  //no paths are found
  null <- Path=[] and makeEmptyList (FreePath)
  or
  (
    //check if the path is free
    null <- request_path_state (Path, PathState) and
    path_free (PathState) then
      null <- assign (Path, FreePath)
  )
  or
  //search for an alternative path which is free
  a (free_path_finder (From, To, FreePath), FRF)
)
```

Figure 8.4: LCC fragment for the “free-path-finder” role

of Figure 7.7 performs two activities: (a) enaction of a separate LCC interaction model in order to get key information on the route state; (b) deduction of the route practicability from the information acquired. Activity (a) is carried out in the case where one information gathering strategy is adopted: the “Check-Route-State” and the “Querier-Reporter” IMs will be respectively enacted in centralized and decentralized scenarios. When the moving peer moves ahead without first checking the route state (no information gathering strategies are adopted), the activity (a) won’t be performed and the route will be assumed to be practicable. Activity (b) will start after completion of the interaction eventually enacted in activity (a) and will usually need the information acquired by the moving peer during such an interaction. The problem of accessing persistent information acquired during the execution of separate interactions is addressed by the OK kernel through the *peer access mechanism* already illustrated in Section 5.2.1, which allows an OKC to access the local knowledge of the peer by invoking methods declared in a specific `PeerAccess` Java class.

Figure 8.5 shows the Java code of the OKC’s method associated to the `request_path_state` constraint that implements the activity (a) mentioned

above. It can be noticed how, depending on the current strategy, the peer either enacts one of two interaction models or sets the route state as “free”. The enactment of a separate interaction model also exploits the peer access mechanism, and specifically takes place by invoking either the *executeIM* method or the *executeIMWithStrategy* method. The latter method differs from the former in that it performs a preliminary filtering of the peers subscribed to the IM to be executed. In particular, before the execution of the “Querier-Reporter” IM, the peer selects a group of reporter peers. More details on this selection mechanism are given later in this section.

```

if (InfoGatheringStrategy.equalsIgnoreCase("no_info")){

    //NO INFORMATION-GATHERING STRATEGY

    pathState = "free";
}
else if (InfoGatheringStrategy.equalsIgnoreCase("centralised")){

    //CENTRALISED INFORMATION-GATHERING STRATEGY

    this.setReceivedPathInfoFromCPU("false");
    String subdescCPInfoIM = "firefighter(" + this.getLocation() + ")";
    invokePeer("executeIM", new ArgumentImpl("role", "path_info_requester"),
              new ArgumentImpl("subscription_desc", subdescCPInfoIM),
              new ArgumentImpl("interaction_desc", "path_info"),
              new ArgumentImpl("accept_policy", "1"),
              new ArgumentImpl("activate_diagnostics", "false"));
}
else{

    //DECENTRALISED INFORMATION-GATHERING STRATEGY

    String subdescPollSensorIM = "querier(" + pathToCheck + ")";
    invokePeer("executeIMWithStrategy", new ArgumentImpl("role", "querier"),
              new ArgumentImpl("subscription_desc", subdescPollSensorIM),
              new ArgumentImpl("interaction_desc", "poll_sensors"),
              new ArgumentImpl("accept_policy", "1"),
              new ArgumentImpl("activate_diagnostics", "false"));
}
}

```

Figure 8.5: Java code for OKC method “*request_path_state*”: interaction model enactment

In what follows, we give some details on both the *centralized control behaviour* and the *decentralized control behaviour*.

Centralized Control Behaviour The centralized scenario is characterized by the presence of the Civil Protection peer who acts as the unique provider

of route state information and relies on reporters, i.e., the main sources of such information. The behaviour of the main actors is the following:

- The **Civil Protection** is subscribed to the *querier* role in the “Querier-Reporter” IM and to the *path_info_provider* role of the “Check_Route_State” IM. It maintains a database of current statuses of locations and answers requests from moving peers for status information;
- Each **Moving Peer** is subscribed to the *emergency-subordinate* role defined in the “Evacuation” IM (see Appendix A.1) and to the *path_info_requester* role defined in the “Check_Route_State” IM. Initially at a given location L , this peer performs the following steps in order to reach the goal destination G :
 1. If $L = G$ then stop
 2. Otherwise:
 - (a) Get one path P from L to G ($P=[P_{head} \mid P_{tail}]$)
 - (b) Check that P is free by interacting with Civil Protection
 - i. If the path is free then
 - A. move from P_{head} to next location L_n
 - B. Back to step (b) with $P=P_{tail}$
 - ii. Otherwise back to step (a) to get an alternative path from L to G
- Each **Reporter** is subscribed to the *reporter* role defined in the “Querier-Reporter” IM. It responds to requests for water level information from a querier (e.g., Civil Protection).

Figure 8.6 schematizes the main interactions between the peers. Full details on “Check_Route_State” and “Querier-Reporter” interaction models are given in Appendixes A.3 and A.4, respectively.

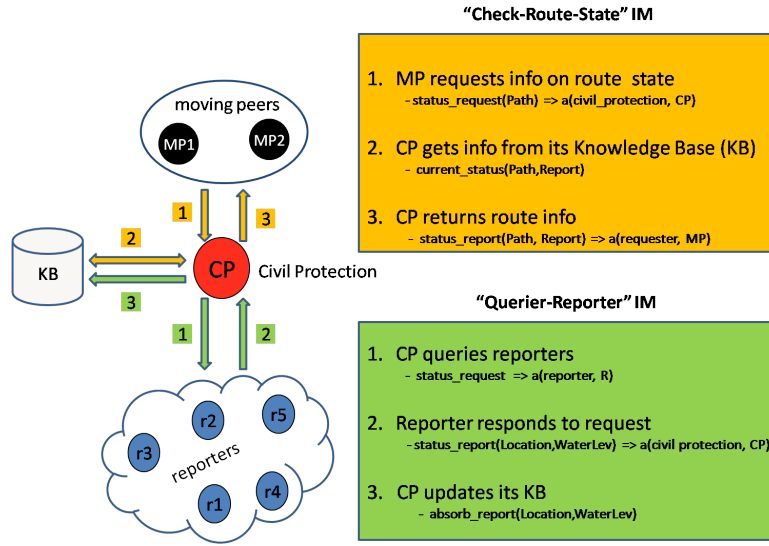


Figure 8.6: Information Gathering: centralized interactions

Decentralized Control Behaviour the decentralized scenario is characterized by the direct interaction between a moving peer and a suitably selected group of reporters. The behaviour of such a peer is reported below:

- Each *Moving Peer* is subscribed to an *emergency-subordinate* role in the “evacuation” interaction (see A.1) and to the *querier* role of the “Querier-Reporter” IM. Initially at a given location L , this peer performs the following steps in order to reach the goal destination G :

1. If $L = G$ then stop
2. Otherwise:
 - (a) Get one path P from L to G ($P=[Phead \mid Ptail]$)
 - (b) Subscribe to the role of querier with subscription description $querier(Ptail)$
 - (c) Choose reporters according to path P
 - (d) Check that P is free by interacting with the selected reporters
 - i. If the path is free then

- Move from P_{head} to next location L_n
 - Back to step (d) with $P=P_{tail}$
 - ii. Otherwise back to step (a) to get an alternative path from L to G
- Each **Reporter** at location N subscribes to the *reporter* role in the “Querier-Reporter” IM with a subscription description of “reporter(N)”. It responds to a request for status information from a querier (e.g., moving peer).

The key point in the section above, is how the moving peer selects a suitable group of reporter peers. Suppose the peer has to move from location L to location G through path $P = [L, A, B, G]$. In this case, A and B represent intermediate locations (or nodes). Assume reporters $R1$, $R2$, $R3$, $R4$ and $R5$ are at nodes A , L , F , B and G respectively and they are subscribed to the *reporter* role as specified above. Figure 8.7 shows the selection process as a sequence of steps. In step 1, the moving peer MP subscribes to the querier role with subscription description $querier(A, B, G)$. This means that it is interested in interacting with only those reporters which are present at the location A, B, G specified. In step 2, MP receives a list R of all reporters subscriptions from the OK Discovery Service. In this example, such list would be $R = [R1(A), R2(L), R3(F), R4(B), R5(G)]$. In step 3, MP selects the reporters of interest, that is, $R1$, $R4$ and $R5$. In step 4, the MP starts interacting via the “Querier-Reporter” IM with the selected reporters (green-bordered circles with blue fill).

8.2.2 The e-Response simulator

The simulator is designed to represent the environment where all the involved agents act. It is composed of three modules which are themselves

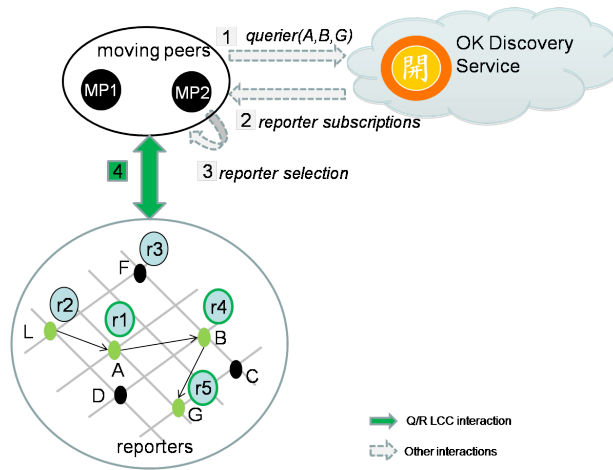


Figure 8.7: Decentralized Information Gathering: selection of reporters

peers: the controller, the flood sub-simulator, and the visualiser (see Figure 3). The controller regulates the simulation cycles and the management of the simulated agent activities; the flood sub-simulator reproduces the actual evolution of the 1966 flood in Trento; the visualiser stores simulation information used by the GUI to view a simulation run in a step-by-step way. The simulator does not interfere or help coordinate peer's actions in the peer network. It is used to simulate the real world.

A full explanation of the interaction models designed to implement the simulator is to be found in Appendix B. The next sections, describe the overall functioning of the simulation engine.

Controller

The controller is the core of the simulator: it drives the simulation cycles and keeps track of the current state of the world. In order to achieve that, it needs to know what changes are happening to the world and updates its state accordingly. After updating its state, it also informs the relevant peers of these changes. The simulation thus evolves through cycles (or time-steps). A simulation cycle foresees two main operations:

- ***Gathering changes***: the controller receives information about the changes that happened to the world: (a) it receives the disaster (e.g., flood) changes from the disaster sub-simulator via the “Flood” IM and (b) it serves requests of performing (move) actions with the “Perform-Action” IM (see Figure 8.2). In this latter interaction, the controller verifies whether certain actions are legal or not before they are performed, and if a certain action is illegal, the peer is informed of the reason of failure;
- ***Informing peers***: the controller sends information about the changes that happened in the world: (a) it sends, at each time-step, local changes to each connected peer via the “Sensory-Info” interaction model and (b) it sends to the visualiser information on - (i) the locations of all connected peers; (ii) the status of the reporter peers (e.g., available, responding to requests) and (iii) the water level registered; here, the “Visualiser” interaction model is used.

Before a simulation cycle starts, some preliminary activities are performed such as: establishing key parameters (e.g., maximum number of simulation cycles, timeouts, water level thresholds), connecting with the flood sub-simulator, sharing with it the initial topology of the world, and adding connecting peers. Once a simulation cycle terminates, the controller updates the time-step and starts the next cycle. Notice that, due to the modularity of the above architecture, it is reasonably easy to add as many disaster sub-simulators (e.g., landslides, earthquake, volcanic eruption, etc.) as needed.

To simulate the afore-mentioned activities, the single interaction model “Simulation-Cycles” is designed⁴. An LCC code snippet is given in Figure 8.8. It only shows the key role of the controller. The constraints specified in

⁴See appendix B.1 for full details on this interaction model.

bold are solved by executing the “Flood”, “Sensory-Info” and “Visualiser” interaction models⁵ respectively.

```

a(info_handler(CurrentTimestep, MaxTimestep, SimSleepTime), SIM) ::
  null <- greater(CurrentTimestep, MaxTimestep)
  or
  (
    null <- gather_info(SimSleepTime) and
           send_info(SimSleepTime) and
           inform_visualiser(CurrentTimestep) and
           getCurrentTimestep(NewTimestep) then
    a(info_handler(NewTimestep, MaxTimestep, SimSleepTime), SIM)
  )

```

Figure 8.8: LCC fragment for the “info-handler” role taken by the controller

Flood sub-simulator

The flood sub-simulator’s goal is to simulate a flood in Trento town (Italy). The equation defined in its core OKC is based on flooding levels and flooding timings resulted from a scientific flood simulation of Trento town, developed by the International Institute for Geo-Information Science and Earth Observation and by the University of Milano-Bicocca [2].

This study is based on a very detailed digital terrain model of the river Adige valley, on historical hydrological data of the flood experienced in Trentino in 1966 and on the localization of ruptures of the river’s dike. It also takes in consideration floodplain topography changes from (the year) 1966 to (the year) 2000 caused by modifications in vegetation spaces, in agricultural regions, in industrial zones, in urban areas and in infrastructures. A two-dimensional finite element flood propagation model is used to reconstruct the 1966 flood and to show how the terrain alterations affects the flood behaviour. This 2-D model, at regular time intervals, generates

⁵The “Flood”, “Sensory-Info” and “Visualiser” IMs are fully explained in Appendixes B.4, B.5 and B.6, respectively.

two maps for both the water height and the flow velocity. Once such maps are created, they are then transformed into five *indicator maps*:

- *Maximum water level*: the maximum water level, in meters, reached by the flood (Figure 8.9);
- *Maximum flow velocity*: the maximum speed (in meters per second) of the water flow (Figure 8.10);
- *Maximum impulse*: the maximum amount of water that has been moved (Figure 8.11);
- *Maximum water level rising speed*: the maximum hourly increase of the water depth, calculated in meters per hour (Figure 8.12);
- *Arrival time of first water*: the time when the flood arrives at a given position (Figure 8.13).

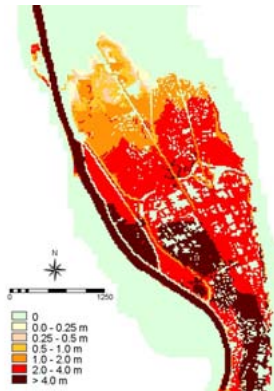


Figure 8.9: Maximum Water Level [2]

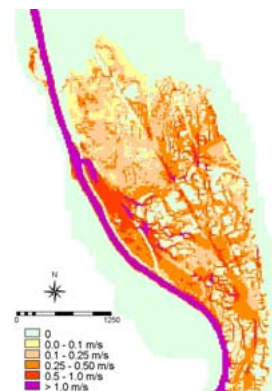


Figure 8.10: Maximum Flow Velocity [2]

To the purpose of our testbed, the territory is divided into flooded areas: each area is characterised by the maximum water height reached during the inundation and the time when this level is touched. These flooded areas are obtained by digitizing the indicator maps of Figures 8.9 and 8.13; To

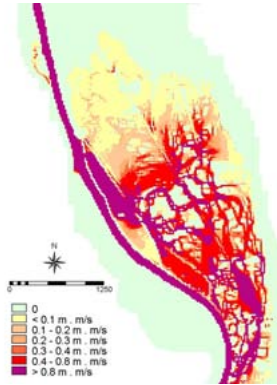


Figure 8.11: Maximum impulse [2]

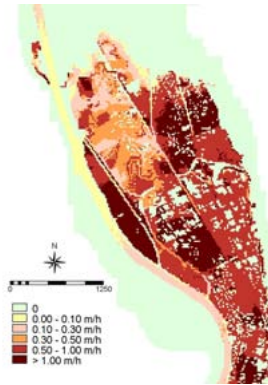


Figure 8.12: Maximum water level rising speed [2]

maintain our simulation realistic but simple, we have assumed that each area reached its maximum flooding level in one hour.

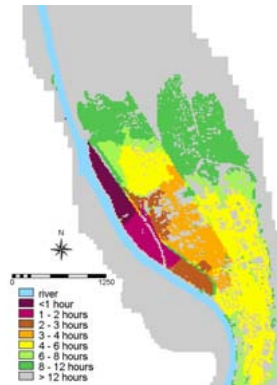


Figure 8.13: Arrival time of first water [2]

Figures 8.14 and 8.15 show a zoom on the north region of Trento town. In particular, they depict the maps of the flooded areas and represent, respectively, the maximum water level and the time when it is reached. Such maps are used in our testbed to create two different tables in a geographical database.

Each table has a field, called **node**, representing x,y coordinates of digitalized points. Moreover, the first table has a field, called **MaxWL** (Maximum Water Level), that is the maximum water height for a node. The second ta-

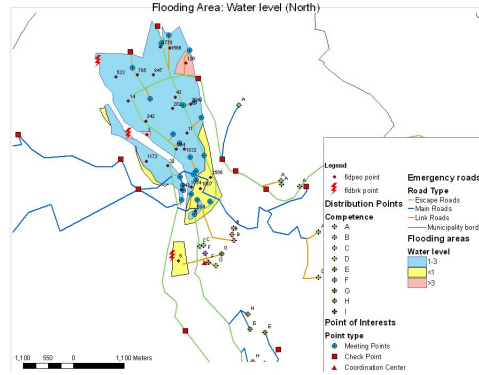


Figure 8.14: Maximum water level in Trento Nord

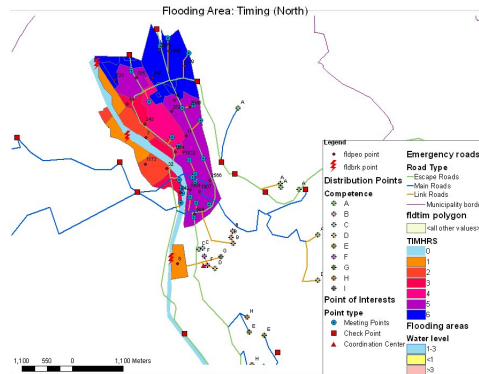


Figure 8.15: Arrival time of maximum water level

ble, instead, has a field, called MT(Maximum Time), that describes the time, in hours, at which the flood reaches the maximum water level at a node. This value is calculated digitalizing the map showing the time arrival of the first water (see Figure 8.13) and making the assumption that the time required to culminate the flood is one hour. Finally, at OK-simulation⁶ time, only the selected data of the topology of the region interested by the current simulation are joined in a single table using an Open Geospatial Consortium standard spatial SQL query.

Given the data stored in the two tables of the geographical database, and assuming that the time required to culminate the flood is one hour, the

⁶We denote our testbed simulation as the “OK-simulation”, in order to distinguish it from the one in [2].

flooding law used during the OK-simulation to calculate the flood changes for a given node at a time-step t is:

$$\left\{ \begin{array}{ll} f(t) = 0 & \text{if } t < (MT - 1) * T \\ f(t + 1) = f(t) + \frac{(MaxWL)}{T} & \text{if } (MT - 1) * T \leq t < MT * T \\ f(t) = f(MT * T) & \text{if } t \geq MT * T \end{array} \right. \quad (8.1)$$

where T is the number of time-steps per hour. It can be noticed that the flood level is 0 from the beginning of the OK-simulation to one hour before MT , i.e., the time at which the flood reaches the maximum level. Second, in an hour the flood increments from 0 to $MaxWL$ and finally it stays to $MaxWL$ until the end of the OK-simulation. The time at which the water level starts to decrement is not considered since the number of hours the flood stays at its maximum level is sufficiently high for the purpose of our simulation.

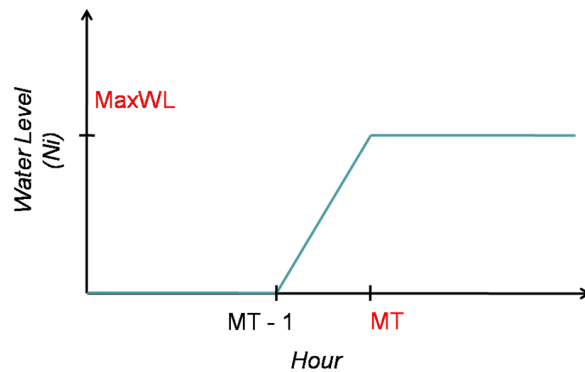


Figure 8.16: Flooding Law

The flood sub-simulator is developed in Java and is fully integrated into the OpenKnowledge kernel. The main component is an OpenKnowledge peer *FloodPeer* that subscribes to two interaction models, i.e., the “Flood

Sub-Simulator_Connection” IM and the “Flood” IM; and stores its core OKC component *FloodSubSimulatorOKC*.

These two interaction models are very simple. The “Flood Sub-Simulator_Connection” IM, explained in detail in Appendix B.2, is enacted just once at the beginning of the simulation by the *connectWithSubSimulators* constraint defined in the “Simulation Cycles” IM. This last IM is described in more detail in Appendix B.1).

The “Flood Sub-Simulator_Connection” IM has two aims:

- The sharing of the topology of the world between the controller and the flood sub-simulator peers;
- The storing, in the controller peer local knowledge, of the connection state of the sub-simulator peer.

The “Flood” IM, described in more detail in Appendix B.4, is used by the controller at each time-step, to get from the flood sub-simulator the changes of the flood level at the nodes in the area interested by the simulation. The core parts of this interaction model are the *floodChanges(Time, Changes)* constraint (in the ‘flood-simulator’ role) and the *updateFloodChanges(Changes)* constraint (in the ‘controller’ role). The first constraint implements the flooding law (8.1); the second one performs an update of the water level of only those nodes which were interested by flood changes during the last time-step.

Visualiser

This component enables the GUI used to visualise the simulation. The visualiser module interacts with the controller via the “Visualiser” IM⁷. At every simulation cycle, the visualiser receives, from the controller module, information on: (i) the current time-step; (ii) for each moving peer

⁷The “Visualiser” IM is described in full details in AppendixB.6

connected, its current geographical position; (iii) for each node, its geographical position, the current water level and the status of the sensor located at that node. Such information are stored in a log file, and then visualized in a GUI.

Figure 8.17 shows the appearance of the GUI at the first time-step of a simulation run. A grey hat represents an emergency subordinate; a green dot represents a reporter peer available for giving information on the water level registered; a grey dot represents a reporter agent giving this information; the water level at a location is depicted as a blue circle, which size depends on how high the water level is. Figure 8.18 shows the evolution over time of the flood in the area at risk.

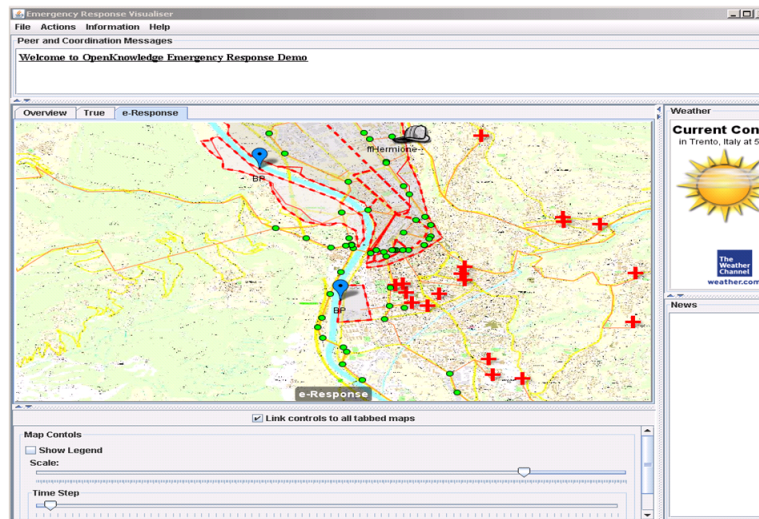


Figure 8.17: Emergency GUI

The simulations are visualized on the GUI in order to analyse the movements of the emergency peers. In this way, we can verify the correct mechanism in the coordination among the agents, and validate the overall interaction-based scenarios.

Figures 8.19 and 8.20 show a simulation run for the centralized and the decentralized scenario respectively. Figure 8.19 shows the agent outside the flooded area. Here, all the dots are grey, meaning that all reporters are

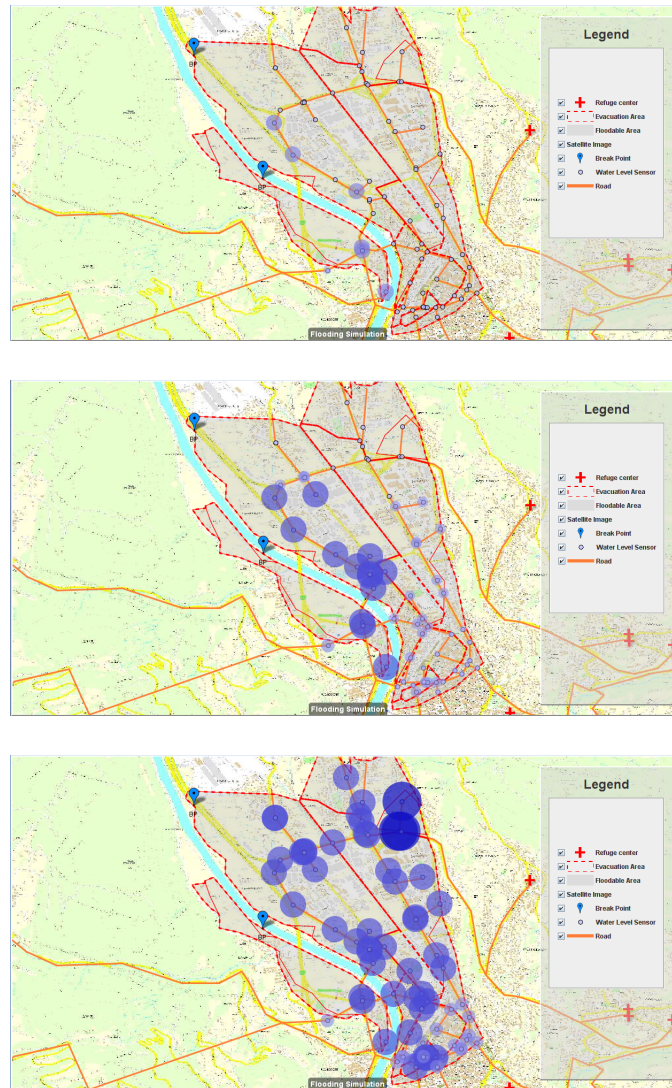


Figure 8.18: Flood Evolution

being queried by the Civil Protection in order to obtain the water level of their location. Some of them register high levels of water.

Figure 8.20 shows the agent moving along a route, which can be deduced by the grey dots ahead the agent; these dots represent in fact those reporters located along the route followed and therefore queried by the moving agent; all the other reporters remain available (green dots).

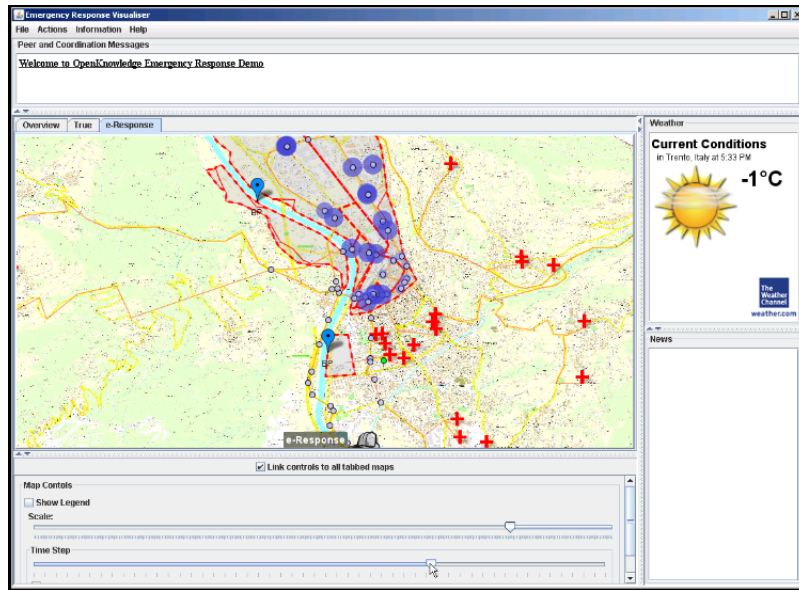


Figure 8.19: GUI: Centralized information gathering

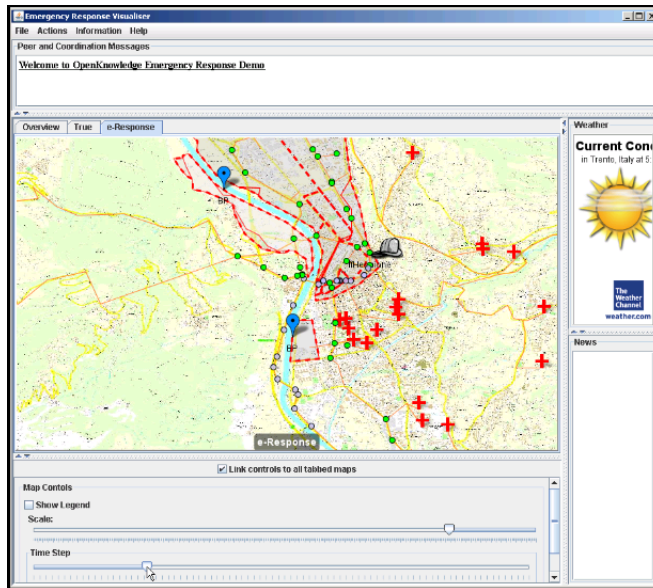


Figure 8.20: GUI: Decentralized information gathering

8.3 OK component reuse

To build the e-Response simulation system described so far, we strongly benefit from the possibility of reusing OK components. In particular, the components reused to implement both centralized and decentralized sce-

narios are LCC specifications and OKC plug-ins.

8.3.1 Reuse of LCC interaction models

Most of the interaction models are reused. However, from the point of view of the information gathering strategy, the key interaction model is the “Querier-Reporter”. It is worth to describe here the mechanism through which this very same specification is used to enable both centralized and decentralized scenarios (see Figure 8.21). Simply, in the centralized scenario the Civil Protection peer subscribes to this IM with the subscription description *querier(all)*. This makes the *CP* peer interacting with all the reporters. Moreover, the *CP* peer enacts the interaction continuously, i.e., at each time-step. On the contrary, in the decentralized scenario the moving peer subscribes to the same interaction with the subscription description *querier(Path)* as already described in Section 8.2.1. Finally, the peer *MP* enacts this interaction only when needed, i.e., when it has to move.

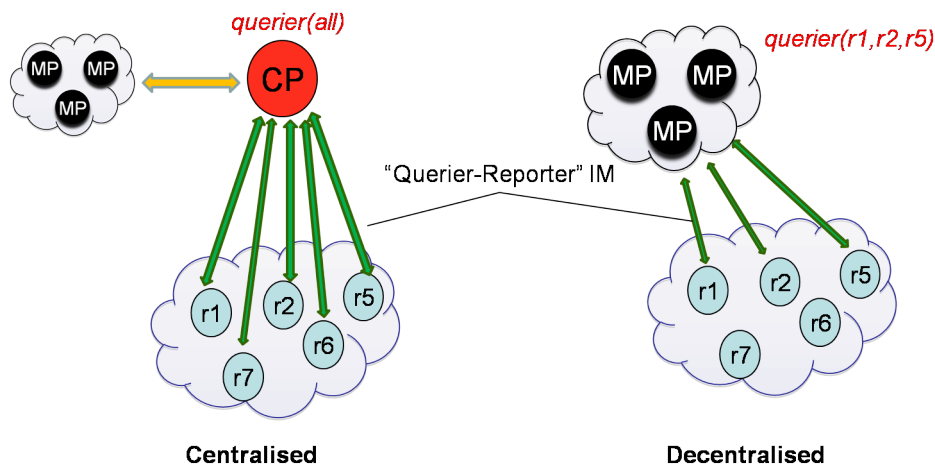


Figure 8.21: The “Querier-Reporter” IM Reuse

8.3.2 Reuse of OKC methods

Besides reusing the interaction models, the e-Response system implementation is aided by the reuse of OKC components, although with a minor degree. From one side, we have different peers using exactly the same OKC. For example, the *UtilOKC* Java class provided by the OK kernel was a useful OKC component shared by all peers in our simulation. Such a component provides basic methods for variable increment, decrement, comparison and so on. On the other side, we have OKC components organized in a hierarchical way. By exploiting the Java's inheritance mechanism it is, in fact, possible to define OKC components which are used by many peer types and, when needed, are extended to yield specific behaviour. For example, the OKC component *ConnectOKC* used to include all the methods needed to solve the constraints in the "Sensory-Info" IM, is stored by the peers previously denoted as *physical peers*; and is extended for each peer so that it can tackle the constraints differently. In our case, while the Civil Protection peer needs to enact the "Querier-Reporter" IM every time it receives sensory info from the controller, the moving peer does not.

Such system will be helpful in many ways: (1) in doing research on interaction-centered coordination models; (2) in providing an evaluation of how appropriate an LCC-like protocol is to reflect dynamic coordinated tasks; (3) to bring the approach into the application field. The evaluation entails two dimensions: (a) it concerns a qualitative validation of the simulation system and the interaction models; such validation can be done by involving the local institutions working in crisis management. Expert people in the field can give useful feedbacks by saying how well the simulations reflect the actual plans. Moreover, the simulations can give useful hints to the experts by showing possible scenarios they didn't foresee, since the plan is actually written on paper and never tried in real situations. In this

case, the simulation system could act as a training system for the experts;

8.4 Conclusion

This chapter has described in detail the realization of a simulation environment fully integrated with the OK system and composed of a network of peers that coordinate with each other through a predefined set of LCC interaction models. This peer-network is integrated with an e-response simulator to constitute an e-Response simulation environment. Such environment is used to validate the interaction-based emergency response scenarios and shows that all parts of the OK system are capable of working cohesively in the desired manner, to support different information-gathering patterns of interaction (i.e., centralized and decentralized strategies).

Part IV

Experimental Evaluation

Chapter 9

Experimental Testbed

The previous chapter described the details of the e-response simulation prototype. The provision of such a prototype is helpful in a number of ways.

This chapter describes how the simulation environment was exploited. In particular, it presents the experimental testbed used to provide a preliminary study on the efficacy of different information-gathering strategies in the e-Response domain; and, to test the OK infrastructure upon which it is built. The chapter is organized as follows: Section 9.1 presents an overview of the testbed; Section 9.2 defines the hypotheses to be tested in order to conduct a preliminary evaluation of the efficacy of different information-gathering strategies; Section 9.3 deals with the set-up of the experiments designed for the above purposes by introducing the experimental performance measures, the experimental variables and the assumptions under which the experiments are performed. Section 9.4 presents and discusses the results obtained and, finally, Section 9.5 concludes the chapter.

9.1 Testbed overview

The developed e-Response testbed mainly consists of an agent-based e-Response simulation environment fully integrated with the OpenKnowl-

edge infrastructure and through which existing emergency plans are modelled and simulated. The e-Response testbed is composed of the following components:

1. Suite of LCC interaction models (ca.13) supporting three different peer coordination strategies: baseline, centralized and decentralized coordination. These strategies has been already discussed in Section 6.2.2;
2. A simulator capable of modeling a flood event in Trentino, using real GIS/flood data (see section 8.2.2);
3. Suite of OKC components (ca. 25) enabling emergency response peers to engage in interactions;
4. Suite of peers (ca. 75) modeling emergency agents;
5. Suite of experiments aimed at both analyzing different information-gathering strategies and testing the OK infrastructure;

In particular, the testbed is conceived to pursue the following objectives:

- Show the OK system in action, illustrating that all parts of the system are capable of working cohesively in the desired manner;
- Provide a preliminary evaluation on how different information gathering strategies impact emergency response tasks;
- Test whether the OK infrastructure fully supports the interaction-based scenarios described.

While the achievement of the first objective depends on an appropriate design of the OK components (IMs, OKCs); the second goal requires to be elaborated at a more concrete level. The next section introduces the hypotheses considered to start investigating this second point. The last point is addressed in Section 9.4.

9.2 Experimental hypotheses

In order to study the effectiveness of centralized and distributed information gathering strategies in emergency response tasks, we setup the following hypotheses:

1. any information-gathering strategy is preferable to a more “blind” strategy in which information are not requested;
2. under ideal conditions (e.g., perfect communication channels, reliable information), centralized and decentralized information-gathering strategies are equally effective to increase the chances of successfully completing emergency response operations;
3. when undesired and/or unexpected conditions (e.g., failures of sensors and/or breakdown of communication channels) arise, a decentralized information gathering strategy is more advantageous over a centralized one, provided that an appropriate supporting infrastructure exists.

Although trivial, the first two hypotheses have to be verified to make the proposed model adherent to the reality.

The following methodological steps are taken to conduct the experimental evaluation:

- establishment of performance measures;
- analysis of the variables involved;
- determination of meaningful assumptions;
- configuration of the experiments;
- experiment execution.

These steps are described in the next sections. In order to verify the hypotheses, it is crucial to run each experiment a significant number of times. This is probably the most timeconsuming and burdensome task since the number of peers involved is considerable and the OK kernel version used is not completely stable. In this thesis, we make few experimental runs to verify the first two hypotheses; the experimentation needed to verify the third hypothesis requires a more reliable underlying communication infrastructure and is left to future work.

9.3 Experimental design

The design of our experiments consists of the establishment of adequate performance measures, an analysis of the variables involved, a set of meaningful assumptions and a configuration of experiments.

9.3.1 Performance measures

The performance measures considered are:

- (a) the *percentage of times* an emergency subordinate arrives at destination;
- (b) the *travelling time*, i.e., the number of time-steps needed to reach the destination.

These indicators are used to compute and compare the experimental results.

9.3.2 Experimental variables

The experimental variables are the following:

- A. *Number of moving peers*: number of peers moving to a specific destination. Since the main aim of the summative experiment is to compare two different strategies (centralized vs decentralized) rather than making a realistic simulation, this variable is fixed to 1 in all experiments. By running an experiment a certain number of times, we compute the performance (a) of the simulated scenario;
- B. *Peer speed*: the speed - measured in km/h - with which a moving peer moves along the paths;
- C. *Distance*: set of paths in the topology connecting two locations. To get significant results it is crucial to consider, for each experiment type, routes covering both safe and flood-prone areas;
- D. *Flooding law*: how the flood evolves over time. The flooding law, which is fixed in our experiments, markedly affects the outcome of an experiment run: a peer may either arrive at destination or be blocked, depending on how rapidly the flood propagates along the route taken;
- E. *Water level threshold*: the water level threshold - expressed in meters - above which a node can be considered blocked;
- F. *Topology*: the set of geographical locations considered and whose status can be reported by some peer;
- G. *Distribution of reporter peers*: the set of nodes where reporter peers are present. In our experiments, this variable is the set of nodes composing only the routes involved in a given experiment; incrementing this number is useful to test the capacity of the OK kernel to support many peers;
- H. *Number of (reporter) peers per node*: the number of reporters located in one node. As above, this variable is useful to test the scalability of

the OK system and, moreover, the effectiveness of some of its modules (e.g., the trust module);

- I. *Degradation of the CP communication channel*: measured as the likelihood of a fault in the communication channel of the CP peer. For example, having a degradation of 80% means to have this peer serving incoming requests only 20% of the times. In particular, by setting this variable, a specific type of fault (channel fault) and its severity can be simulated;
- J. *Degradation of reporter communication channels*: defines, for all reporter channels, the probability of their disruption. For example, having a degradation of 30% means to have each reporter peer serving incoming requests with the likelihood of 70%. This variable plays a role in the experiments which foresee the presence of channel fault conditions. As for the previous parameter, the setting of this variable determines the degree of severity of the channel fault;
- K. *Distribution of trustworthy (reporter) peers*: defines the number of reporter peers having a trustworthy behaviour, that is, peers which always report accurate water level values. It is expressed as the percentage over the total number of reporter peers. This variable plays a role in the experiments which foresee the presence of fault conditions. In particular, by setting this variable, a specific type of fault (fault due to inaccurate info), its location and its severity can be simulated. In the implemented experiments, we assume all peers are trustworthy.

We have reported above the whole list of variables defined in the design of the experiments. In our evaluation testing, we have used a sub-set of the listed variables.

9.3.3 Assumptions

In order to contextualize and interpret properly the results, it is important to explicitly list the assumptions made in the simulation in order to simplify it, namely:

They are:

- the CP peer has infinite resources (under ideal conditions). This means that the peer is able to serve any number of simultaneous requests and the communication channel never breaks. Therefore, under this assumption, bottleneck problems due to overwhelming requests never occur;
- a querier, asking a certain number of reporters for an info, will receive all the answers within a time-step. This is due to how the time-step interval is set: the value is such that the time elapsing between one time-step and the next one is sufficiently high to guarantee the replies from all the reporters.

With such assumptions, we simulate a scenario where the performances of both centralized and decentralized communication infrastructures are comparable.

9.3.4 Experiment configuration

The configuration of the experiment is sketched in Table 9.1: three types of experiments are foreseen. In the first type of experiment, an emergency subordinate moves towards its assigned destination without gathering any information; the other two types are related to the centralized and decentralized information-gathering strategies. Each experiment is run 10 times; at each run, the only variables that change are the destination assigned (variable C) and the nodes where reporters are present (variable

G). The flooding law, the peer speed, the number of moving peers and the number of reporters located at each node, remain unchanged during all runs.

The results obtained from the simulation runs are described in the next section.

			Variable Settings							
Exp. Type	IG	Runs	A	B	C	D	E	F	G	H
1	none	10	1	30	1 distance x run	fixed	0.3	fixed	1 config x run	1
2	centralized	10	1	30	1 distance x run	fixed	0.3	fixed	1 config x run	1
3	decentralized	10	1	30	1 distance x run	fixed	0.3	fixed	1 config x run	1

Table 9.1: Experiments configuration

9.3.5 Experiment execution

In order to run an experiment, a number of processes equal to the number of total agents considered need to be launched. For this purpose, a Java program reads a selected configuration of parameters and then launches the processes. Furthermore, the use of bash scripts allows to run an arbitrarily large number of contiguous, independent simulations. This mechanism exploits the distributed nature of the OK platform. For example, while the DDS is run in one server, the processes associated with the reporter peers are launched in a different machine, the other emergency response peers in a third one.

9.4 Results

This section shows and discusses the results obtained from running the simulations. In particular, subsection 9.4.1 presents the preliminary evaluation of the efficacy of the two information-gathering strategies; subsection 9.4.2 deals with the evaluation of the OK infrastructure, which is made possible by running the same experiments.

9.4.1 Information gathering strategies

We conducted the experiments described in the previous section. Independently of the kind of strategy adopted, the final goal of a moving peer is to safely reach the assigned destination. We consider the following possible situations that might occur:

- (1) the agent reaches the destination by following the first route found;
- (2) the agent proceeds along a path, finds blocked nodes but finally reaches the destination by taking an alternative path;
- (3) the agent doesn't reach the destination at all.

We refer to each situation as the *outcome* of the experiment.

Figure 9.1 shows the percentage - over the total number of experimental runs - of times a moving peer arrives at destination, depending on the IG strategy adopted. As it can be seen from the figure, the adoption of centralized or decentralized strategies increases the chances to arrive to the final destination. Although not surprising by themselves, these results confirm that the simulations are coherent with the expectations, hence, with the trivial hypotheses.

Figure 9.2 shows the outcome distribution obtained by running the second experiment 10 times. As can be seen, 80% of the times, the experiment has outcome (1) (the peer reaches the destination without problems) while 20% of the time, the outcome is (3) (the peer does not reach the destination). The outcome (2) is never obtained. Although we setup the routes in order to cover different kind of areas (either safe or flood-prone areas), the case where an agent finds free routes after a re-routing never happens. This could be explained by considering how the design of the flooding law and its related "flood speed" affects the evolution of the scenario. The outcome

distribution related to the second experiment, which simulates the decentralized scenario, is identical to the one found for the first experiment and hence is not reported here. This result can be explained with the assumptions previously made: asking information on the route's practicability to either the *CP* peer or reporters scattered around the city does not make the difference.

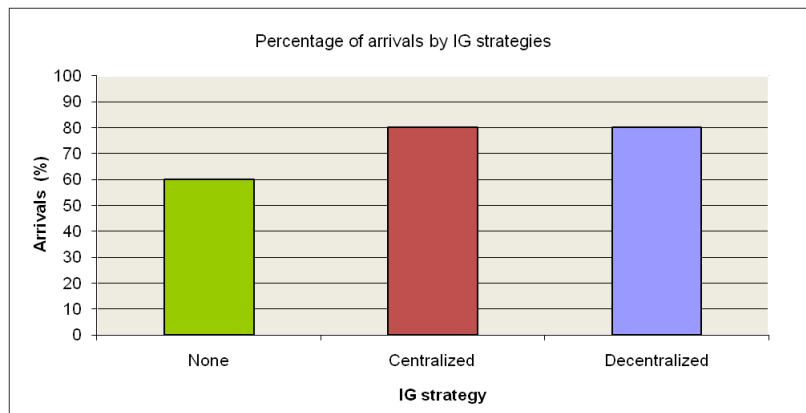


Figure 9.1: Percentage of arrivals by IG strategies

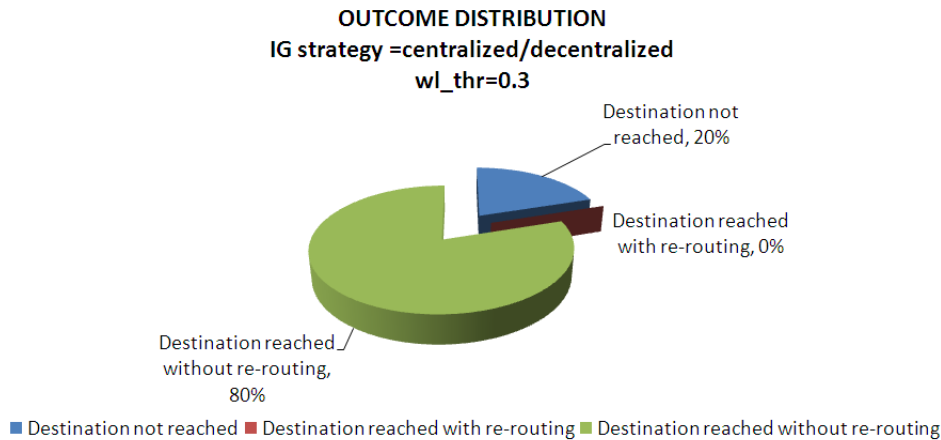


Figure 9.2: Outcome Distribution (centralized/decentralized scenario)

Figure 9.3 shows the time taken (measured as the number of simulation time-steps) by an agent to reach the goal location according to the shortest distance (in terms of intermediate locations) between the initial position

and the final destination. The trend is shown for both experiments. It can be observed that, in both cases, the time needed to achieve the goal is nearly equal to the shortest distance. This can be explained by the way the simulation is designed - an agent moves from a location to the next one exactly in one time-step - and by the missing outcome (2). Finally, Figure 9.3 reveals very similar trends for both centralized and decentralized scenarios. Again, this is mainly due to the assumptions made and the variable settings.

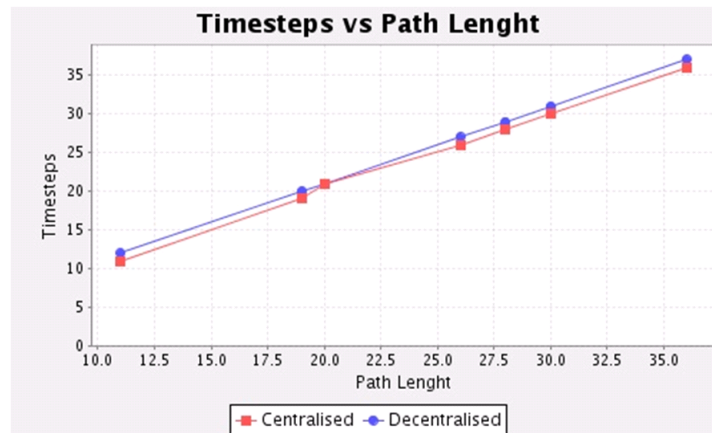


Figure 9.3: Time-steps vs. Path Length

In view of the results described above, we can conclude that our simulations adhere to what we would expect in reality; and that under the selected - ideal - assumptions, centralized and decentralized information-gathering strategies are comparable.

9.4.2 The OK infrastructure

As mentioned earlier, the conducted experiments served also as a black-box functional testing of the underlying OK infrastructure. In particular, we determined the percentage of times the OK infrastructure failed in supporting the whole simulation process, and in which stage; also, we

determined in which phase of the coordination mechanism (e.g., interaction subscription, interaction run), failures occurred.

Figure 9.4 shows the percentage of times that the simulation process failed to complete.

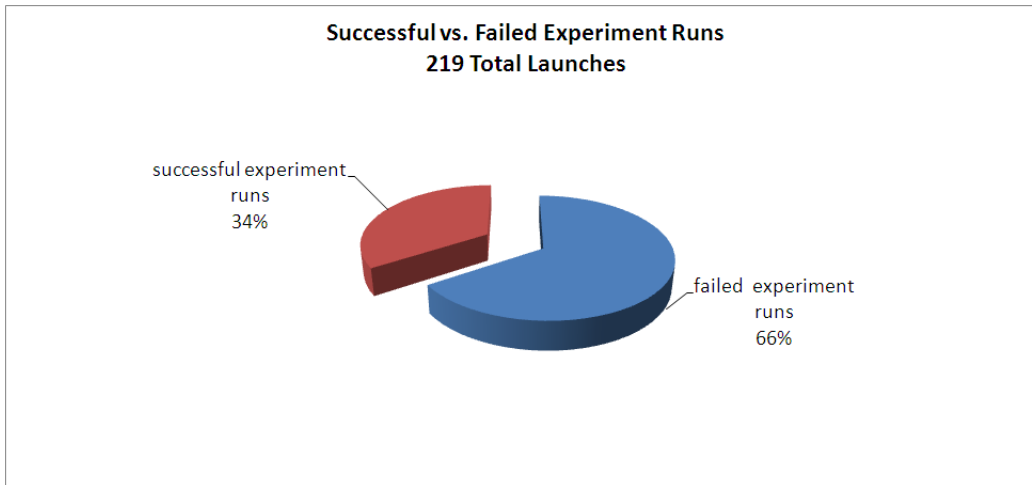


Figure 9.4: Successful vs. Failed Experiment Runs

Figure 9.5 depicts the distribution of the type of failure. As the figure shows, 99% of the failures are due to problems in the subscription phase of a peer to an IM; only 1% of the simulations are stopped during the interaction run phase.

As revealed during the development phase of the OK kernel, the subscription problem is due to a malfunctioning in the OK discovery service: a query reply is never returned to the peer which is looking for an IM (step 3 of Figure 5.7). However, the reason why this happens is not known. Due to the complexity of our test-case, it is difficult to devise a simpler scenario that consistently reproduces the issue; thus, to get more insights in the subscription problem, we conducted a further analysis.

It is worth to mention here how a simulation run is performed: the processes associated with the reporter peers are launched first; the reporter peers subscribe to the “reporter” role of the “Poll-Sensor” IM. Note that

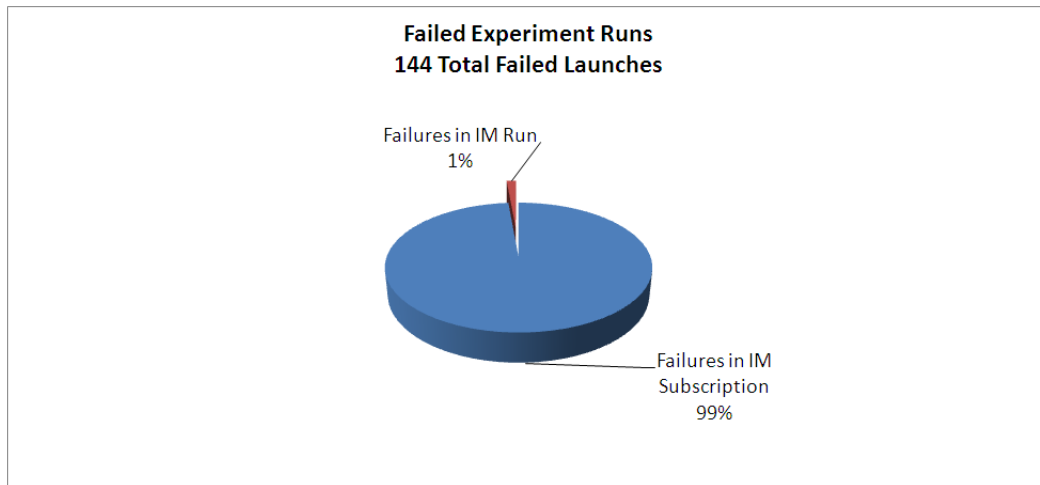


Figure 9.5: Failed experiment runs

in this phase of the simulation run, no interactions are being run since the other peers processes are not launched yet. When all reporter peers are subscribed, the processes associated with the other e-response peers are launched so that the whole simulation can start. In summary, there are two phases in a simulation run: (a) subscription phase of the reporter peers (no interactions are executed); (b) the actual simulation where multiple, interleaved interactions are executed.

Figure 9.6 illustrates the distribution of the subscription failures. As the figure shows, 30% of the subscription failures occur during the first phase of the simulation run, i.e., when no interactions are executed; 70% of the subscription failures mainly occur during the execution of the interactions.

The fact that most of the subscription failures occur in the second phase of the simulation run, let one think that the problem might be connected to the many IMs enacted during the execution of other interactions, hence, to the complexity of the test case in terms of IMs. To further investigate this, we derived the diagram of Figure 9.7, which gives a more fine-grained view of the subscription failures by distinguishing the type of IMs for which the failures occur. The figure shows that subscription failures occur in

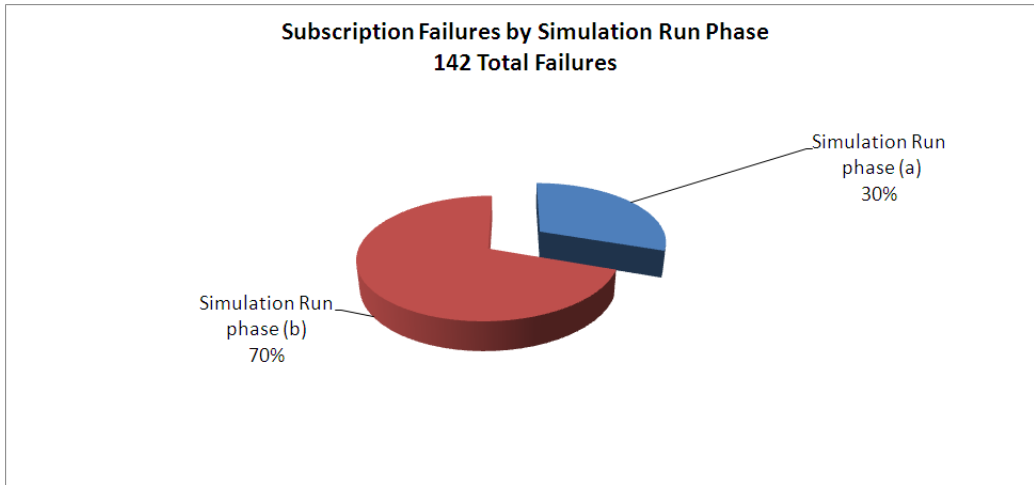


Figure 9.6: Subscription failures by simulation run phase

a low percentage (i.e., 1%) for some interactions (i.e., evacuation, peer connection). These interactions are not enacted within the execution of other IMs and are not executed repeatedly; therefore, at a very first look, it seems that there is a link between the subscription problem and the enactment of multiple interleaved interactions.

However, these are preliminary results which must be taken carefully and further validated; nevertheless, they could provide a direction where to investigate further this kind of failure issue.

9.5 Conclusion

This chapter has presented a testbed which is used to conduct initial analysis on the efficacy of different information-gathering strategies in e-response settings; and, to test the OK underlying supporting infrastructure. In particular, preliminary results show that our simulations adhere to realistic scenarios, and that under ideal conditions centralized and decentralized information-gathering strategies are comparable.

In relation to the testing of the OK infrastructure, our complex scenario

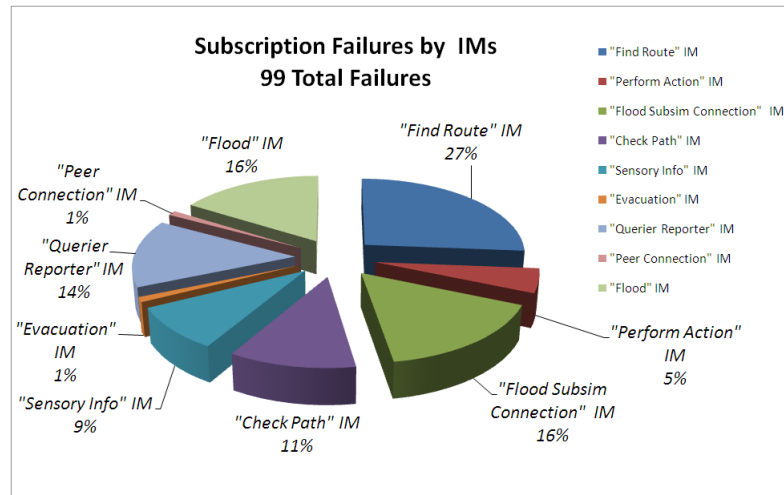


Figure 9.7: Subscription failures by IMs

has shown some limitations with respect to the coordination mechanism. In particular, failures in the subscription phase of the interaction lifecycle have emerged. With the aim to investigate whether this kind of failure is connected with the complexity of the simulation, we conducted an initial analysis on the type of IMs for which the subscription failures occur. Although preliminary and limited, the findings suggest that the discovery mechanism of the OK infrastructure is weak in supporting multiple interleaved interactions.

Part V

Final Discussions

Chapter 10

Related Work

In this thesis, we apply interaction-oriented frameworks to information sharing within complex contexts such as emergency response. We adopt simulations to establish whether specific software systems (e.g., the OK system) realizing such frameworks support emergency response activities, and to study how different information-gathering strategies impact on such activities.

Many research works exist which adopt simulations as means to evaluate the effectiveness of different strategies, and the suitability of IT solutions for crisis response management [16, 74, 101, 14].

In [16], a multi-agent based approach is described to model and simulate the dynamics of large scale crisis situations. As in our work, the model and the simulator are based on official rescue plans and on realistic data. However, the simulator is based on a multi-agent platform which does not provide mechanisms to discover, and gather together heterogeneous peers.

The DrillSim simulation system has been developed to analyze the effect of information technology solutions in emergency response [74]. It is a multi-agent system where each agent simulates a person. The core component of DrillSim is the simulation engine: it simulates the geographic space, the evacuation scenario and the current status of the disaster. The

prototype has a GUI that enables humans to control the simulation and to interact with virtual agents.

The DEFACTO system [101], provides a multi-agent simulation system which is based on a scalable architecture. It allows responders to interact with the coordinating agent team in a complex environment and includes 3D visualization. This simulation environment is used to analyze the impact of teamwork interaction strategies.

Other related research works are either specifically devised for the emergency management area or focused more on the architectural aspect. In particular, FireGrid¹, CASCOM², and WORKPAD³ are among such projects.

In the FireGrid project [57], a system to help fire-fighters in e-Response activities has been built which adopts Grid-techniques. The scenario considered here is an indoor environment, where real-time sensor data are collected, processed and presented to support responders in decision making.

In the CASCOM project (Context-Aware Business Application Service Coordination in Mobile Computing Environments) an intelligent agent-based peer-to-peer (Ip2p) environment was developed [58]. Here, the service coordination mechanism relies on Semantic Web technologies, such as OWL-S and WSMO, rather than on explicit lightweight protocols.

The WORKPAD project aims at designing and developing an innovative software infrastructure (software, models, services, etc.) for supporting collaborative work of human operators in emergency/disaster scenarios [80]. A set of front-end peer-to-peer communities providing services to human workers, mainly by adaptively enacting processes on mobile ad-hoc networks, is part of the system developed [32]. Each community is lead by a super-peer, which is the only peer managing workflow composition and

¹<http://www.firegrid.org>

²<http://www.ist-cascom.org>

³<http://www.workpad-project.eu>

coordination in an adaptive manner.

Chapter 11

Conclusions

Coordination technologies play a crucial role to support effective interactions among processes, whether reactive (e.g., web services) or proactive (e.g., autonomous agents). In the past few years, interaction-oriented frameworks have been proposed, which enable a priori unknown agents to engage in coordination activities thanks to the sharing of interaction models specified in executable protocol languages. Software systems have started to be developed to apply such frameworks to concrete use.

In particular, the OpenKnowledge framework has been proposed as such an interaction-oriented framework and the OpenKnowledge system has been developed for its realization. In the OpenKnowledge system, interaction models specified in the Lightweight Coordination Calculus (LCC) are shared first-class entities which enable distributed peers to engage in coordination tasks. Although the realization of the OpenKnowledge approach is promising, its application in complex and dynamic scenarios, is still a challenge.

This thesis work aims at tackling this challenge, by applying the OpenKnowledge framework to realistic contexts such as emergency response.

The choice of crisis response as a target domain is driven by its potential to deal with a distributed-knowledge and dynamic environment. Here, the

adoption of a distributed infrastructure such as the one provided by OpenKnowledge, can reveal to be effective in dealing with a flexible coordination of emergency response activities.

The main contribution of this thesis is to have applied the OpenKnowledge framework in realistic contexts. In particular, the research has been carried out to:

- design LCC interaction protocols to model complex emergency response coordination tasks;
- provide an agent-based simulation environment as a means to informally validate interaction models relative to different information-gathering strategies in emergency contexts;
- fully use and test a distributed infrastructure enabling and executing interaction protocols such as OpenKnowledge;
- provide a preliminary experimental evaluation of the efficacy of different information gathering strategies in emergency response settings.

Modular and reusable LCC protocols have been designed to model relatively complex emergency response scenarios. Such design has affected the development of the OK infrastructure, thus enriching it with features needed to bring the approach towards the application field.

Part of an emergency plan has been modeled in terms of LCC specifications and a simulation environment has been built to informally validate the designed interaction-based scenarios. The simulation architecture is composed of an agent network and a core engine which allows agents to coordinate with each other via predefined LCC interactions.

The developed simulation environment is fully integrated with the OK system and has been used to test its capability to support different models of information-sharing. Results have shown that, the OK infrastructure

is able to support complex coordination tasks; however, some limitations have appeared in relation to the subscription phase of the coordination mechanism.

Finally, the simulation system has been exploited to carry out a preliminary analysis on the effectiveness of centralized and decentralized information sharing strategies in emergency contexts. An experimental testbed is built to this purpose. Preliminary results show that our simulations adhere to realistic scenarios, and that under ideal conditions centralized and decentralized information-gathering strategies are comparable.

The research conducted has thus provided a contribution towards the application of interaction-based approaches to information sharing within realistic, emergency response scenarios.

Chapter 12

Future Work

Some limitations, due to both the functionalities of the framework and the features of the LCC protocol language, were experienced during the development of the simulator. More specifically:

1. Interaction models are constructed manually since there are no supporting tools. This prevented an easy and fast development process. Also, since LCC requires a logic background, the system could not currently be used by end-users to simulate their own processes;
2. The LCC protocol does not directly provide a way to compose hierarchies of interaction models. It is not possible to refer to a “sub-interaction model” from a “main interaction model”. Some technical solutions were applied that allowed to bypass the problem but still the specification lost in clarity. In fact, to analyse how different interaction models are called and nested, it is necessary to look at how the decision procedure of a single agent process is implemented. This raises interesting questions as to where an interaction protocol “ends” and the agent process “starts”;
3. According to the OpenKnowledge framework, once an interaction starts, the participants are fixed and new comers cannot enter it.

This prevented us to simulate extremely realistic emergency scenarios where it is likely to have peers suddenly leaving and joining. For example, if such functionality was provided, it could have been possible to deal with an unknown number of peers. However, in this case, we would have encountered a limitation of the protocol language: the lack of a parallel operator which prevents to send a message, in multicast, to all those peers playing a given role.

The issue described in point 1, has previously been raised by Walton et al. [120, 121] for the MAP protocol language, analogous to LCC. Here different approaches are considered to solve the problem: provision of a graphical tool, automatic generation of protocols by means of a planning process, extension of the protocol language thus to make it more suitable for p2p architectures.

Recently, Besana and Barker [19] extended the LCC language and introduced the “scene” and the “parallel” operators: the first abstracts an interaction model and the second allows for real multicast. We could apply this extended version of the protocol to our interaction-based scenarios, to investigate whether this helps in a faster and easier protocol design process.

A further observation regards the flexibility of the protocol (as it was used in the case study considered) to model realistic scenarios. On one hand, LCC interaction protocols are rigid in the sense that they specify predefined “workflows” of agent-oriented services. On the other hand, they provide flexibility at least in two ways:

- By means of external decision procedures: they provide an interface between the dialogue protocol and the rational process of the agents;
- By allowing the participants to dynamically change the protocol itself during the coordination task.

In our research we explored the first characteristic, which allows to separately design arbitrary complex agents. However, engineering more proactive agents would have increased the dynamicity of the interactions. In this case, a decision procedure in an LCC interaction model might have represented the goal needed by the proactive process to plan a sequence of actions in order to achieve that goal.

In our research, we explored the first characteristic, which allows to separately design arbitrary complex agents. However, engineering more proactive agents would have increased the dynamicity of the interactions. In this case, a decision procedure in an LCC-like interaction model could represent the goal to be achieved by a proactive agent, and could be implemented by deriving a suitable plan, i.e. a sequence of actions, to achieve that goal.

Let's make a concrete example of how this could work in the context of an emergency situation: suppose an emergency coordinator peer asking its team members to perform the specific task of moving to a precise location. The agent acting on behalf of a team member might first deliberate the best way to find the path: rely on its own knowledge? Ask a predisposed route service? Consult agent peers in its vicinity? In this last case, the problem is how to consult the peer colleagues. A planning process could be enacted to provide the agent in need with a suitable sequence of actions (e.g., messages to be sent to specific neighbor peers). Such sequence of actions could be synthesized in an interaction model. This interaction model would be nested in the previous one, that is, would be enacted at the point of the decision procedure presents in the original interaction model, the one which started the whole coordination task; moreover, it might be even shared with other peers willing to solve similar problems.

From the discussion above, we can deduce that a real flexibility of process coordination can be achieved by effectively designing both interaction

models and agent-oriented processes. Investigating more deeply the trade-off between the complexity of the interaction models and that of the agent processes could lead to an interesting direction.

Finally, the exploitation of the second feature outlined above, the one for which an LCC protocol can be adapted while it is executed, would certainly guarantee an extreme flexibility. Along this line, it would be interesting to apply the findings of McGinnis [78] to the concrete scenario considered in this thesis.

Regarding the use of the e-response simulation prototype, we started with a preliminary evaluation on the impact of different information-gathering strategies in the response phase of an emergency. We could complete the evaluation and find out when a decentralized information-gathering strategy is to be preferred to a centralized one, hence, to evaluate the opportunity to adopt or switch to a peer-to-peer solution for emergency management.

In regards to the evaluation of the underlying infrastructure, the same environment can be used to evaluate whether the use of a peer-to-peer framework - as the one provided by OpenKnowledge - improves on conventional centralized systems when specific fault conditions arise.

Finally, another direction for future work is the improvement of the simulation experiments: Web interfaces could be designed which allow the input of parameters to ease the simulation run process.

Bibliography

- [1] J. Abian, M. Atencia, P. Besana, L. Bernacchioni, D. Gerloff, S. Leung, J. Magasin, A. Perreau de Pinninck, X. Quan, D. Robertson, M. Schorlemmer, J. Sharman, and C. Walton. *OpenKnowledge Deliverable 6.3: Bioinformatics Interaction Models*, 2008.
- [2] Dinand Alkema, Angelo Cavallin, Mattia De Amicis, and Andrea Zanchi. Valutazione degli effetti di un alluvione: il caso di trento. *Studi Trentini di Scienze Naturali : Acta Geologica*, 78:55–62, 2003.
- [3] H. Aminoff, B. Johansson, and J. Trnka. Understanding coordination in emergency responses. In *Proc. EAM Human Decision-Making and Manual Control*, pages 1111–1128, 2007.
- [4] A Arkin, S Askary, B Bloch, Y Goland, N Kartha, C K Liu, S Thatte, P Yendluri, and A Yiu. Web services business process execution language. 2004.
- [5] A Arkin, S Askary, S Fordin, W Jekeli, K Kawaguchi, D Orchard, S Pogliani, K Riemer, S Struble, P Takacsi-Nagy, I Trickovic, and S Zimek. Web service choreography interface (wsci). Technical report, W3C, 2002.
- [6] A. Artikis and J. Pitt. Specifying open agent systems: A survey. *Engineering Societies in the Agents World IX*, pages 29–45, 2009.

- [7] A. Artikis, M. Sergot, and J. Pitt. Specifying electronic societies with the Causal Calculator. *Agent-Oriented Software Engineering III*, pages 1–15, 2003.
- [8] J.L. Austin, J.O. Urmson, and M. Sbisà. *How to do things with words*. Harvard Univ Pr, 1975.
- [9] X. Bai and David Robertson. Service choreography meets the web of data via micro-data. In *In Proceedings of the AAAI Spring Symposium on Linked Data Meets Artificial Intelligence (LINKEDAI 2010)*. AAAI Press, 2010.
- [10] Adam Barker, Paolo Besana, David Robertson, and Jon B. Weissman. The benefits of service choreography for data-intensive computing. In *CLADE '09: Proceedings of the 7th international workshop on Challenges of large applications in distributed environments*, pages 1–10, New York, NY, USA, 2009. ACM.
- [11] Adam Barker, Christopher D. Walton, and David Robertson. Choreographing web services. *In IEEE Transactions on services computing*, 2(2), 2009.
- [12] A. Barros, M. Dumas, and P. Oaks. A critical overview of the web services choreography description language. *Business Process Trends White Paper*, 2005.
- [13] A. Barros, M. Dumas, and A.H.M. Ter Hofstede. Service interaction patterns. *Business Process Management*, pages 302–318, 2005.
- [14] I. Becerra-Fernandez, M. Prietula, R. Valerdi, G. Madey, D. Rodriguez, and T. Wright. Design and development of a virtual emergency operations center for disaster management research, training,

- and discovery. In *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, page 27. IEEE, 2008.
- [15] Saoud N Bellamine-Ben, J Dugdale, B Pavard, and M Ben Ahmed. Towards planning for emergency activities in large-scale accidents: An interactive and generic agent-based simulator. In *Proceedings of the first International workshop on Information Systems for Crisis Response and Management*, 2004.
- [16] Saoud N Bellamine-Ben, T Ben Mena, J Dugdale, B Pavard, and M Ben Ahmed. Assessing large scale emergency rescue plans: an agent based approach. special issue on emergency management systems. *International Journal of Intelligent Control and Systems*, 11:260–271, 2006.
- [17] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. Jade-a white paper. *EXP in search of innovation*, 3(3):6–19, 2003.
- [18] Federico Bergenti, César Cáceres, Alberto Fernández, Nadine Fröhlich, Heikki Helin, Oliver Keller, Ari Kinnunen, Matthias Klusch, Heimo Laamanen, António Lopes, Sascha Ossowski, Heiko Schuldt, and Michael Schumacher. Context-aware service coordination for mobile e-health applications. In *ECEH*, pages 119–130, 2006.
- [19] Paolo Besana and Adam Barker. An executable calculus for service choreography. In *OTM Conferences (1)*, pages 373–380, 2009.
- [20] Paolo Besana, David Dupplaw, and Adrian De Pinnick. Openknowledge deliverable 1.3: Plug-in component architecture. Technical report, OpenKnowledge, 2007.
- [21] Paolo Besana, Fiona McNeill, Fausto Giunchiglia, Lorenzino Vaccari, Gaia Trecarichi, and Juan Pane. Web service integration via

- matching of interaction specifications. Technical report, University of Trento, Dipartimento di Ingegneria e Scienza dell'Informazione, 2008.
- [22] Paolo Besana, Adam Patkar V. Barker, David Robertson, and David Glasspool. Sharing choreographies in openknowledge: A novel approach to interoperability. *Journal of Software, Special Issue on Semantic Extensions to Middleware*, 4(8), 2009.
- [23] A. Bucchiarone and S. Gnesi. A survey on services composition languages and models. In *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, pages 51–63, 2006.
- [24] Churton Budd. Information Gathering and Technology Use During Disaster Deployments.
- [25] Paul A Buhler, José M Vidal, and Harko Verhagen. Adaptive workflow = web services + agents. In *ICWS*, pages 131–137, 2003.
- [26] L. Cabac and D. Moldt. Formal semantics for AUML agent interaction protocol diagrams. *Agent-Oriented Software Engineering V*, pages 47–61, 2005.
- [27] G Casella and V Mascardi. From auml to ws-bpel. Technical report, Computer Science Department, University of Genova, 2001.
- [28] Amit K. Chopra, Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. Reasoning about agents and protocols via goals and commitments. In *AAMAS '10: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*, pages 457–464, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems.

- [29] R. Cost, Y. Chen, T. Finin, Y. Labrou, and Y. Peng. Using colored petri nets for conversation modeling. *Issues in Agent Communication*, pages 178–192, 2000.
- [30] R.S. Cost, Y. Chen, T. Finin, Y. Labrou, and Y. Peng. Modeling agent conversations with colored petri nets. 1999.
- [31] R.S. Cost, Y. Labrou, and T. Finin. Coordinating agents using agent communication languages conversations. In *Coordination of Internet agents*, page 196. Springer-Verlag, 2001.
- [32] Massimiliano De Leoni, Fabio De Rosa, and Massimo Mecella. Mobidis: A pervasive architecture for emergency management. In *WET-ICE*, pages 107–112, 2006.
- [33] A Perreau De Pinninck, D Dupplaw, S Kotoulas, M Schorlemmer, R Siebes, and C Sierra. *OpenKnowledge Deliverable 1.2: Peer to peer coordination protocol*, 2006.
- [34] Adrian Perreau De Pinninck, David Dupplaw, Spyros Kotoulas, and Ronny Siebes. The openknowledge kernel. *International Journal of Applied Mathematics and Computer Sciences (IJAMCS)*, 4(3):162–167, 2007.
- [35] L.P. De Silva, M. Winikoff, and W. Liu. Extending agents by transmitting protocols in open systems. In *Proceedings of the Workshop on Challenges in Open Agent Systems, the Second International Joint Conference on Autonomous Agents & Multi-Agent Systems (AAMAS-03), Melbourne, Australia*. Citeseer, 2003.
- [36] G. Decker, M. Kirov, J.M. Zaha, and M. Dumas. Maestro for Lets Dance: An Environment for Modeling Service Interactions. *BPM Demo Session 2006*, page 32.

- [37] G. Decker, O. Kopp, F. Leymann, and M. Weske. BPEL4Chor: Extending BPEL for modeling choreographies. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 296–303. IEEE, 2007.
- [38] G. Decker, H. Overdick, and J.M. Zaha. On the Suitability of WSCDL for Choreography Modeling. *Proceedings of Methoden, Konzepte und Technologien für die Entwicklung von dienstebasierten Informationssystemen (EMISA 2006), Hamburg, Germany, 2006.*
- [39] N. Desai, A. Mallya, A. Chopra, and M. Singh. OWL-P: a methodology for business process development. *Agent-Oriented Information Systems III*, pages 79–94, 2006.
- [40] N. Desai and M.P. Singh. A modular action description language for protocol composition. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 22, page 962. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.
- [41] Sukumar Dwarkanath and Denis Gusty. Information sharing: A strategic approach. In *Proceedings of the 7th International Conference on Information Systems for Crisis Response and Management (ISCRAM)*, 2010.
- [42] M. Esteva, J. Padget, and C. Sierra. Formalizing a language for institutions and norms. *Intelligent Agents VIII*, pages 348–366, 2002.
- [43] M. Esteva, W. Vasconcelos, C. Sierra, and J. Rodriguez-Aguilar. Verifying norm consistency in electronic institutions. In *Proceedings of the AAAI-04 workshop on agent organizations: theory and practice*, pages 8–14, 2004.

- [44] Marc Esteva, David De La Cruz, and Carles Sierra. Islander: an electronic institutions editor. In *AAMAS*, pages 1045–1052, 2002.
- [45] T. Finin and Y. Labrou. Kqml as an agent communication language. In Bradshaw, editor, *Software Agents*, pages 291–316. 1997.
- [46] FIPA. Fipa interaction protocol library specification. 2001.
- [47] FIPA. Foundation for intelligent physical agents. *Communicative act library specification*, 2001.
- [48] I. Foster and C. Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann, 2004.
- [49] L. Fredlund. Implementing ws-cdl. In *Proceedings of the second Spanish workshop on Web Technologies (JSWEB 2006)*, 2006.
- [50] A. Garcia-Camino, P. Noriega, and J.A. Rodriguez-Aguilar. Implementing norms in electronic institutions. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 667–673. ACM, 2005.
- [51] F. Giunchiglia, M. Yatskevich, and F. McNeill. Structure preserving semantic matching. In *Proceedings of the ISWC+ ASWC International workshop on Ontology Matching (OM)*, pages 13–24, 2007.
- [52] Fausto Giunchiglia, Fiona McNeill, and M Yatskevich. Web service composition via semantic matching of interaction specifications. Technical report, DISI, University of Trento, 2006.
- [53] Fausto Giunchiglia, Fiona McNeill, Mikalai Yatskevich, Juan Pane, Paolo Besana, and Pavel Shvaiko. Approximate structure-preserving semantic matching. In *Proceedings of the 7th Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE)*, pages 1234–1237, 2008.

- [54] M. Greaves, H. Holmback, and J. Bradshaw. What is a conversation policy? *Issues in Agent Communication*, pages 118–131, 2000.
- [55] Li Guo, David Robertson, and Yun-Heh Chen-Burger. A generic multi-agent system platform for business workflows using web services composition. In *IEEE Intelligent Agent Technology*, pages 301–307, 2005.
- [56] Li Guo, David Robertson, and Yun-Heh Chen-Burger. A novel approach for enacting the distributed business workflows using bpel4ws on the multi-agent platform. In *ICEBE*, pages 657–664, 2005.
- [57] Liangxiu Han, Stephen Potter, George Beckett, Gavin Pringle, Stephen Welch, Sung-Han Koo, Gerhard Wickler, Asif Usmani, José L. Torero, and Austin Tate. Firegrid: An e-infrastructure for next-generation emergency response support. *J. Parallel Distrib. Comput.*, 70(11):1128–1141, 2010.
- [58] H Helin, M Klusch, A Lopes, A Fernandez, M Schumacher, H Schuldt, F Bergenti, and A Kinnunen. Context-aware business application service co-ordination in mobile computing environments. In *Proceedings of the fourth conference of Autonomous Agents and Multi Agent systems - Workshop on Ambient Intelligence - Agents for Ubiquitous Computing*, 2005.
- [59] T. Iwao, Y. Wada, M. Okada, and M. Amamiya. A framework for the exchange and installation of protocols in a multi-agent system. *Cooperative Information Agents V*, pages 211–222, 2001.
- [60] S. Jain and C.R. McLean. Modeling and simulation for emergency response. In *Workshop Report, Relevant Standards and Tools, National Institute of Standards and Technology Internal Report, NISTIR-7071*, volume 24, 2003.

- [61] B. Johansson. *Joint control in dynamic situations*. Dept. of Computer and Information Science, Linköping universitet, 2005.
- [62] Z. Kang, H. Wang, and P.C.K. Hung. WS-CDL+ for web service collaboration. *Information Systems Frontiers*, 9(4):375–389, 2007.
- [63] Zuling Kang, Hongbing Wang, and Patrick C K Hung. Ws-cdl+: An extended ws-cdl execution engine for web service collaboration. In *ICWS*, pages 928–935, 2007.
- [64] K.M. Khalil, M. H. Abdel-Aziz, M. T. Nazmy, and A. M. Salem. Multi-agent crisis response systems design requirements and analysis of current systems. In *In Proceedings of the 4th International Conference on Intelligence Computing and Information Systems*, 2009.
- [65] S. Kotoulas and R. Siebes. Adaptive routing in structured peer-to-peer overlays. In *3rd Intl. IEEE workshop on Collaborative Service-oriented P2P Information Systems (COPS workshop at WETICE07), Paris, France, IEEE Computer Society Press, Los Alamitos*. Citeseer, 2007.
- [66] Spyros Kotoulas and Ronny Siebes. Deliverable 2.2: Adaptive routing in structured peer-to-peer overlays. Technical report, OpenKnowledge, 2007.
- [67] Y Labrou, T Finin, and Y Peng. The current landscape of agent communication languages. *Intelligent Systems*, 14:45–52, 1999.
- [68] F. Lin, D. Norrie, W. Shen, and R. Kremer. A schema-based approach to specifying conversation policies. *Issues in agent communication*, pages 193–204, 2000.

- [69] Maurizio Marchese, Lorenzino Vaccari, Gaia Trecarichi, Nardine Osman, and Fiona McNeill. Interaction models to support peer coordination in crisis management. In *Proceedings of the 5th International Conference on Information Systems for Crisis Response and Management (ISCRAM)*, pages 230–241, 2008.
- [70] Maurizio Marchese, Lorenzino Vaccari, Gaia Trecarichi, Nardine Osman, Fiona McNeill, and Paolo Besana. An interaction-centric approach to support peer coordination in distributed emergency response management. *Special Issue on Intelligent Decision Making in Dynamic Environments: Methods, Architectures and Applications of the Intelligent Decision Technologies (IDT): An International Journal*, 3(1), 2009.
- [71] Maurizio Marchese, Lorenzino Vaccari, Gaia Trecarichi, Pavel Shvaiko, Juan Pane, Nardine Osman, and Fiona McNeill. Openknowledge deliverable 6.7: Interaction models for eResponse. Technical report, OpenKnowledge, 2008.
- [72] David Martin, Massimo Paolucci, Sheila Mcilraith, Mark Burstein, Drew Mcdermott, Deborah McGuinness, Bijan Parsia, Terry Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia Sycara. Bringing semantics to web services: The owl-s approach. In *First Int. Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, pages 26–42, 2005.
- [73] F.J. Martin, E. Plaza, J.A. Rodríguez-Auilar, et al. Jim-a java interagent for multi-agent systems. In *In Proceedings of the AAAI-98 Workshop on Software Tools for Developing Agents*, 1996.
- [74] D Massaguer, V Balasubramanian, S Mehrotra, and N Venkatasubramanian. Multi-agent simulation of disaster response. In *Proceedings*

- of the First International Workshop on Agent Technology for Disaster Management*, 2006.
- [75] N. Maudet and B. Chaib-Draa. Commitment-based and dialogue-game-based protocols: new trends in agent communication languages. *The Knowledge Engineering Review*, 17(02):157–179, 2002.
- [76] J. McGinnis and T. Miller. Amongst first-class protocols. *Engineering Societies in the Agents World VIII, LNAI*, 2007.
- [77] J. McGinnis and D. Robertson. Dynamic and distributed interaction protocols. In *In Proceedings of the AISB 2004 Convention*, pages 45–54, 2004.
- [78] J. McGinnis and D. Robertson. Realizing agent dialogues with distributed protocols. *Agent Communication*, pages 106–119, 2005.
- [79] J. McGinnis, D. Robertson, and C. Walton. Using distributed protocols as an implementation of dialogue games. In *Presented EUMAS*. Citeseer, 2003.
- [80] M Mecella, T Catarci, M Angelaccio, B Buttazzi, A Krek, S Dustdar, and G Vetere. Workpad: an adaptive peer-to-peer software infrastructure for supporting collaborative work of human operators in emergency/disaster scenarios. In *Proceedings of the 2006 International Symposium on Collaborative Technologies and Systems*, 2006.
- [81] T. Miller and P. McBurney. Using constraints and process algebra for specification of first-class agent interaction protocols. In *Proceedings of the 7th international conference on Engineering societies in the agents world VII*, pages 245–264. Springer-Verlag, 2007.
- [82] Tim Miller, Peter McBurney, Jarred McGinnis, and Kostas Stathis. First-class protocols for agent-based coordination of scientific instru-

- ments. *Enabling Technologies, IEEE International Workshops on*, 0:41–46, 2007.
- [83] M. Nowostawski, M. Purvis, and S. Cranefield. A layered approach for modelling agent conversations. In *Proceedings of the Second International Workshop on Infrastructure for Agents, MAS and Scalable MAS, the Fifth International Conference on Autonomous Agents*, pages 163–170, 2001.
- [84] J. Odell, H.V.D. Parunak, and B. Bauer. Extending UML for agents. *Ann Arbor*, 1001:48–103, 1999.
- [85] J. Odell, H. Van Dyke Parunak, and B. Bauer. Representing agent interaction protocols in UML. In *Agent-Oriented Software Engineering*, pages 201–218. Springer, 2001.
- [86] Juan Pane, Carles Sierra, Gaia Trecarichi, Maurizio Marchese, Paolo Besana, and Fiona McNeill. *OpenKnowledge Deliverable 4.9: Summative report on GEA, trust and reputation: integration and evaluation results*, 2008.
- [87] C. Peltz. Web services orchestration and choreography. *Computer*, pages 46–52, 2003.
- [88] C. Perrow. *Normal accidents: Living with high-risk technologies*. Princeton Univ Pr, 1999.
- [89] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. *Artificial Intelligence: Methodology, Systems, and Applications*, pages 106–115, 2004.
- [90] J. Rao and X. Su. A survey of automated web service composition methods. *Semantic Web Services and Web Process Composition*, pages 43–54, 2005.

- [91] C. Reed. Dialogue frames in agent communication. In *Multi Agent Systems, 1998. Proceedings. International Conference on*, pages 246–253. IEEE, 2002.
- [92] D. Robertson. A lightweight coordination calculus for agent systems. *Declarative agent languages and technologies II*, pages 183–197, 2005.
- [93] David Robertson. A lightweight method for coordination of agent oriented web services. In *Proceedings of AAAI Spring Symposium on Semantic Web Services, California, USA*, 2004.
- [94] David Robertson. Multi-agent coordination as distributed logic programming. In *Proceedings for International Conference on Logic Programming*, 2004.
- [95] David Robertson, Adam Barker, Paolo Besana, Alan Bundy, Yun-Heh Chen-Burger, David Dupplaw, Fausto Giunchiglia, Frank Van Harmelen, Fadzil Hassan, Spyros Kotoulas, David Lambert, Guo Li, Jarred McGinnis, Fiona McNeill, Nardine Osman, Adrian Perreau de Pinninck Bas, Ronny Siebes, Carles Sierra, and Christopher D. Walton. Models of interaction as a grounding for peer-to-peer knowledge sharing. *LNCS Advances in Web Semantics*, 2007.
- [96] David Robertson, Fausto Giunchiglia, Frank Van Harmelen, Maurizio Marchese, Marta Sabou, Marco Schorlemmer, Nigel Shadbolt, Ronald Siebes, Carles Sierra, Christopher D. Walton, Srinandan Das-mahapatra, David Dupplaw, Paul Lewis, Mikalai Yatskevich, Spyros Kotoulas, Adrian De Pinninck, and Antonis Loizou. Open knowledge semantic webs through peer-to-peer interaction. Technical report, 2006.
- [97] Dumitru Roman, Uwe Keller Holger Lausen, Jos De Bruijn, Rubén Lara, Michael Stollberg, Alex Polleres, Dieter Fensel, and Christoph

- Bussler. Web service modeling ontology (WSMO). *Applied Ontology*, 1(1):77–106, 2005.
- [98] S. Ross-Talbot. Orchestration and Choreography: Standards, Tools and Technologies for Distributed Workflows. In *NETTAB Workshop-Workflows management: new abilities for the biological information overflow*, 2005.
- [99] S Ross-Talbot and T Fletcher. Web services choreography description language: Primer. Technical report, W3C, 2006.
- [100] B.T.R. Savarimuthu, M. Purvis, M. Purvis, and S. Cranefield. Agent-based integration of web services with workflow management systems. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 1345–1346. ACM, 2005.
- [101] Nathan Schurr, Janusz Marecki, J. P. Lewis, and Milind Tambe. The defacto system: Training tool for incident commanders. In *In IAAI*, 2005.
- [102] J.R. Searle. *Speech acts: An essay in the philosophy of language*. Cambridge university press, 1970.
- [103] R. Siebes, D. Dupplaw, S. Kotoulas, A.P. De Pinninck, F. Van Harmelen, and D. Robertson. The openknowledge system: an interaction-centered approach to knowledge sharing. In *Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS-Volume Part I*, pages 381–390. Springer-Verlag, 2007.
- [104] M. Singh. A social semantics for agent communication languages. *Issues in agent communication*, pages 31–45, 2000.

- [105] M. Singh. Agent communication languages: Rethinking the principles. *Communications in Multiagent Systems*, pages 37–50, 2003.
- [106] Munindar P Singh, Amit K. Chopra, Nirmal Desai, and Ashok U Mallya. Protocols for processes: programming in the large for open systems. *SIGPLAN Not.*, 39:73–83, 2004.
- [107] E Sirin, B Parsia, D Wu, J A Hendler, and D S Nau. Htn planning for web service composition using shop2. *J. Web Sem.*, 1:377–396, 2004.
- [108] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated discovery, interaction and composition of semantic web services. *Journal of Web Semantics*, 1(1):27–46, 2003.
- [109] A. Tate. The helpful environment: Geographically dispersed intelligent agents that collaborate. 2006.
- [110] R.K. Thiagarajan, A.K. Srivastava, A.K. Pujari, and V.K. Bulusu. BPML: A process modeling language for dynamic business models. 2002.
- [111] Gaia Trecarichi, Veronica Rizzi, Maurizio Marchese, Lorenzino Vaccari, and Paolo Besana. Enabling information gathering patterns for emergency response with the openknowledge system. *Special Issue on Advanced Data Mining in Ubiquitous Environment, Computing and Informatics Journal*, 29(4), 2010.
- [112] Gaia Trecarichi, Veronica Rizzi, Lorenzino Vaccari, Maurizio Marchese, and Paolo Besana. Openknowledge at work: exploring centralized and decentralized information gathering in emergency contexts. In *Submitted to the 6th International Conference on Information Systems for Crisis Response and Management (ISCRAM)*, 2009.

- [113] Gaia Trecarichi, Veronica Rizzi, Lorenzino Vaccari, Juan Pane, and Maurizio Marchese. Openknowledge deliverable 6.8: Summative report on use of ok approach in eResponse: integration and evaluation results. Technical report, OpenKnowledge, 2008.
- [114] M. Turoff. Past and future emergency response information systems. *Communications of the ACM*, 45(4):29–32, 2002.
- [115] Lorenzino Vaccari, Maurizio Marchese, Fausto Giunchiglia, Fiona McNeill, Stephen Potter, and Austin Tate. Openknowledge deliverable 6.5: Emergency monitoring scenarios. Technical report, OpenKnowledge, 2006.
- [116] Lorenzino Vaccari, Maurizio Marchese, and Pavel Shvaiko. Openknowledge deliverable 6.6: Emergency response gis service cluster. Technical report, OpenKnowledge, 2007.
- [117] C. Walton. Protocols for web service invocation. In *In Proceedings of the AAI Fall Symposium on Agents and the Semantic Web (ASW05)*, 2005.
- [118] C. Walton. Uniting agents and web services. *AgentLink News, Issue 18*, pages 26–28, 2005.
- [119] C.D. Walton. *Agency and the semantic web*. Oxford University Press, USA, 2007.
- [120] Christopher Walton and Adam Barker. An agent-based e-science experiment builder. In *In Proceedings of the 1st International Workshop on Semantic Intelligent Middleware for the Web and the Grid*, 2004.
- [121] Christopher D. Walton. Typed protocols for peer-to-peer service composition. In *P2PKM*, 2005.

- [122] D.N. Walton and E.C.W. Krabbe. *Commitment in dialogue: Basic concepts of interpersonal reasoning*. State Univ of New York Pr, 1995.
- [123] P. Yolum and M. Singh. Commitment machines. *Intelligent Agents VIII*, pages 235–247, 2002.
- [124] P. Yolum and M.P. Singh. Flexible protocol specification and execution: applying event calculus planning using commitments. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, pages 527–534. ACM, 2002.
- [125] J. Zaha, A. Barros, M. Dumas, and A. ter Hofstede. Lets dance: A language for service behavior modeling. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 145–162, 2006.
- [126] M. Zahiri and M. Khayyambashi. An agent-oriented executive model for service choreography. *Journal of Theoretical and Applied Information Technology (JATIT)*, 14(2), 2010.

This appendix includes technical details for the interaction models used to model the evacuation phase. In the subsequent LCC code snippets: (i) a constraint which is solved by enacting a separate interaction model is specified in bold; (ii) the comments are preceded by the character string “//”.

Appendix A

Peer Network Interaction Models

In what follows, we describe the interaction models used by the emergency peers in the selected use case, i.e. the evacuation plan.

A.1 Evacuation.lcc

This interaction model represents the main one to simulate the evacuation phase. It can be used in all those situations where an emergency chief sends the directive of reaching specific locations to its subordinates. It foresees the main roles *emergency_chief* and *emergency_subordinate* which, in our simulation, are played by a fire-chief and a fire-fighter peer respectively (see LCC code below).

The role of the emergency chief is simply that of retrieving a list of available subordinates (*getPeers*¹ constraint), assigning a destination to each subordinate (*assign_goal* constraint) and sending an *alert* message containing the destination to her/him.

The emergency subordinate, denoted as “moving peer” from now on, receives the above message from the chief and prepare to satisfy the directive. The constraints *set_goal* and *get_current_position* are solved in order

¹This constraint is not defined by the designer of the interaction models but is already provided in the OK kernel

to set the goal to be achieved (reach the goal destination G) and get the current position $CurrPos$. The role *goal_achiever* is then taken. The activities of the emergency subordinate thus evolve through three roles: the afore mentioned *goal_achiever* role which abstractly models the activity of searching for a path and moving towards the goal; the *free_path_finder* role which defines the operations needed to find a free path; the *goal_mover* role which models the actions needed to move towards the goal destination.

```

r(emergency_chief,initial)
r(emergency_subordinate,necessary)
r(goal_achiever,auxiliary)
r(free_path_finder,auxiliary)
r(goal_mover,auxiliary)

a(emergency_chief,FFC)::
  null <- getPeers("emergency_subordinate", FFL) then
  a(emergency_chief(FFL),FFC)

a(emergency_chief(FFL),FFC) ::

  null <- FFL = []

  or

  (
    alert(G) => a(emergency_subordinate,FFL_H)
    <- FFL=[FFL_H|FFL_T] and assign_goal(FFL_H,G) then
    a(emergency_chief(FFL_T),FFC)
  )

a(emergency_subordinate,FF)::

  alert(G) <= a(emergency_chief,FFC) then
  null <- set_goal(G) and get_current_position(CurrPos) then
  a(goal_achiever(CurrPos,G),FF)

```

The *goal_achiever* role is specified with the parameters *From* and *To* which respectively indicate the location from where a peer starts moving and the final destination to be reached (see LCC code below). The comments in the code clearly explicate the logic and meaning of the role.


```
a(goal_achiever(From,To),GA)::  
  
(  
  //moving peer already at destination  
  null <- equal(To,From) and setGoalAchieved(To)  
  
  or  
  
  ( //try to find a free path  
    a(free_path_finder(From,To,FreePath), GA) then  
  
      //no free paths between From and To  
      null <- FreePath=[] and setGoalUnreachable(To)  
  
      or  
  
      //move towards the goal destination along the free path found  
      a(goal_mover(From,To,FreePath),GA)  
    )  
  )  
)
```

The *free_path_finder* role is specified with the input parameters *From* and *To* already mentioned and produces, once it is ended, the output parameter *FreePath* which contains the shortest free path connecting the nodes *From* and *To*. The constraint *find_path* enacts the interaction model “Find-Route” (see next section) in order to find an existing path. This operation is repeated till a free path is found or there are no paths anymore. A free path is a path that is not blocked by the flood. The information on the blockage state of a path are acquired by solving the constraint *request_path_state* which enacts the interaction model “Check-Route-State” (see section A.3 for more details).

```
a(free_path_finder(From,To,FreePath), FRF) ::  
  
null <- find_path(From,To,Path) then  
(  
  //no paths are found  
  null <- Path=[] and makeEmptyList(FreePath)  
  
  or  
  
  (  

```

```

//check if the path is free
null <- request_path_state(Path,PathState) and
  path_free(PathState) then
  null <- assign(Path,FreePath)
)

or

//search for an alternative path which is free
a(free_path_finder(From,To,FreePath), FRF)
)

```

Finally, the role *goal_mover* is used to actually move towards the goal destination. The role consists in moving step by step, from a node to the next one. At every step, the moving peer tries to perform the “move” action (*try_move_action* constraint) as explained in section A.2 with the aim to arrive at the next location along the path. Also, once such location is reached, the peer checks the blockage state of the remaining path through the constraint *request_path_state*². If the peer is prevented to make even the first step, most probably an inaccurate signaling by part of the Civil Protection (CP) happens during the execution of the “Check-Route-State” interaction model; this because the *goal_mover* role is entered only if a free path is found. The logic and meaning of the role just described is made explicit by the comments of the LCC code below.

```

a(goal_mover(Start,Goal,Path), GM) ::

null <- getSubGoal(Path,SubGoal) and
  try_move_action(Start,SubGoal,ActionState) then
(
  //The moving peer has moved
  null <- action_performed(ActionState) and
    update_current_position(SubGoal) then

  //the moving peer has reached the final location
  null <- equal(SubGoal,Goal) and setGoalAchieved(Goal)

or

```

²Notice that a path which was found to be free the first time it was checked, can get blocked subsequently.

```
(
//the moving peer reaches an intermediate node (SubGoal) in the Path
null <- notEqual(SubGoal,Goal) then
//check the blockage state of the remaining path
//and take decision on whether to move
null <- update_path(Path,RestPath) and
request_path_state(RestPath,PathState) and
take_move_decision(PathState,MoveDecision) then

    (// the moving peer proceeds: path is free
    null <- go_for_move(MoveDecision) then
    a(goal_mover(SubGoal,Goal,RestPath), GM)
    )

    or

    //the moving peer stops: path is blocked
    //find alternative free paths from SubGoal to Goal
    a(goal_achiever(SubGoal,Goal),GM)
)
)

or

(
//The moving peer stops: wrong info from CP peer received (fault case)
//find alternative free paths from Start to Goal
null <- update_blocked_nodes(Start,SubGoal) then
a(goal_achiever(Start,Goal),GM)
)
)
```

A.2 Find-Route.lcc

This interaction model is used to retrieve a route connecting two given locations. Two roles are involved: the *route_finder* role, played by an emergency subordinate in our case, and the *route_service* role, taken by a route provider.

The route finder initiates the interaction by sending a *route_request* message to the route service. The message contains the following parameters:

- *PeerName*: the name identifying the requester;
- *From*: the starting location;

- *To*: the final destination;
- *Vehicle*: the means of transport used to move;
- *BlkNodes*: a list of (already known) inaccessible locations which are to be excluded from the path requested.

Once the above message is sent and the reply received with the *route* message, the path *Path* specified in it is stored in the peer local knowledge (*store_path* constraint).

The route service, after reception of the *route_request* message, solves the constraint *get_route* in order to compute the shortest path (*Path*) between the given locations which does not pass by the nodes specified in the list *BlkNodes*. If no such path is found, the parameter *Path* becomes an empty list. In any case, the *route* message is sent with the parameter specified. The LCC code of the interaction just illustrated follows:

```

r(route_finder, initial)
r(route_service, necessary)

a(route_finder, RF) ::
  route_request (PeerName, From, To, Vehicle, BlkNodes) => a(route_service (RS))
  <- get_peer_name (PeerName) and get_current_position (From) and
  get_final_destination (To) and set_vehicle (Vehicle) and
  get_blocked_nodes (BlkNodes) then

  route (From, To, Path) <= a(route_service (RS)) then
    null <- store_path (Path)

a(route_service, RS) ::

  route_request (PeerName, From, To, Vehicle, BlkNodes) <= a(route_finder, RF) then
  route (From, To, Path) => a(route_finder, RF)
  <- get_route (PeerName, From, To, Vehicle, BlkNodes, Path)

```

A.3 Check-Route-State.lcc

This interaction model is used to verify the conditions of the roads and, therefore, the ability for all drivers to be able to arrive at destination. It

involves a peer asking for the blockage status of a given route and a peer providing such kind of information. In our simulation, the requesting peer is a fire-fighter and the info provider is the Civil Protection (CP).

A fire-fighter initiates the interaction by taking the role *path_info_requester* (see LCC code below). Here, the peer first verifies its connection to the simulation and acquires the parameter *WaitTime* which will be used in the next role. After getting the path *Path* to check (*get_path_to_check* constraint), the message *path_info_request* is sent to the info provider. The role *path_info_receiver* is then assumed. In such role, the requesting peer waits for the reply till either this is received or the maximum await time (*WaitTime*) has elapsed. The actions taken when a reply is not received are not foreseen by this interaction; rather, they are established in the requester peer's OKCs. In our simulation, when the peer doesn't obtain the seeked information, the path is considered as it was practicable. When received, the reply is constituted by the *path_info* message and its parameters:

- *BlkNodes*: a list of locations in the requested path which are unreachable;
- *FreeNodes*: a list of locations in the requested path which are reachable;
- *Timesteps*: a list of time-steps relative to the above parameters. Each time-step represents the last time at which the status of the corresponding location has been updated. This parameter allows the requesting peer to know "how old" the searched information is.

The role *path_info_provider* taken by the CP is very simple and consists in receiving the message *path_info_request* and serving the request. Notice that, to serve many requests, the CP peer subscribes to this interaction model at the beginning of the simulation, with an acceptance policy of "all". The request is handled by getting the current time-step

CurrTimestep and obtaining the path info. In our simulation, a local database³ is consulted for this purpose: the constraint *get_path_status* retrieves, if present, the water level registered at the locations specified in the path *Path* at the time *CurrTimestep*. Depending on the water level value, the relative location is inserted in either the *BlkNodes* or the *FreeNodes* list. If the status of a given location exists but refers to a previous time-step, this time-step is considered and put in the list *Timesteps*.

```

r(path_info_requester, initial)
r(path_info_receiver, auxiliary)
r(path_info_provider, necessary, 1)

a(path_info_requester, PIR)::
null <- connected() and getWaitTime(1, WaitTime) then
  path_info_request(Path) => a(path_info_provider(PIP)
    <- get_path_to_check(Path) then
      a(path_info_receiver(WaitTime, 1), PIR)

a(path_info_receiver(WaitTime, N), PIR)::

  null <- equalZero(N)

  or

  (
    path_info(BlkNodes, FreeNodes, Timesteps) <= a(path_info_provider, PIP) then
      null <- store_path_info(BlkNodes, FreeNodes, Timesteps) and
        dec(N, NewN) then
          a(path_info_receiver(WaitTime, NewN), PIR)
  )

  or

  null <- equalZero(WaitTime)

  or

  (
    null <- sleep(1000) and dec(WaitTime, NewWaitTime) then
      a(path_info_receiver(NewWaitTime, N), PIR)
  )

```

³The knowledge base of the CP is filled with information gathered periodically from the reporter peers by means of the “Querier-Reporter” interaction model (see next section for more details).

```
a(path_info_provider, PIP)::  
  
  path_info_request(Path) <= a(path_info_requester, PIR) then  
  
    path_info(BlkNodes, FreeNodes, Timesteps) => a(path_info_requester, PIR)  
      <- getTimestep(CurrentTimestep) and  
         get_path_status(Path, CurrentTimestep, BlkNodes, FreeNodes, Timesteps)
```

A.4 Querier-Reporter.lcc

This interaction used to model the communication between the Unit (CP) and a reporter peer is composed by two main roles: the *querier* role and the *reporter* role (see LCC code below).

The heading of the specification defines that the *reporter* role can be played by more than one peer, the maximum number allowed being 200. The CP takes the role of *querier* with a subscription description of the type “querier(all)”, while a reporter peer subscribes to the *reporter* role with a subscription description of the type “reporter(*node*)”, where *node* univocally identifies a specific geographical location.

By subscribing as a “querier(all)”, the CP specifies that it is interested in all the nodes present in the emergency area. However, if the interest is just in a subset of such locations, it is possible to subscribe as a “querier(*node1*,...,*nodeN*)”. This sort of mechanisms allows a flexible use of the interaction specification which doesn’t need to be modified when the locations of interest change.

The *querier* role entails two sub-roles: the *sender* and the *receiver* role. The CP first gets the current time-step *Timestep* and then retrieves the list of all the peers which are playing the *reporter* role. Notice that these peers can be selected according to one of the three strategies described in [86]. After this, the CP enters the role *sender* in order to send the

message *request_flood_status(Timestep)* to all the selected reporter peers. Once the messages are sent, the *CP* computes a waiting time *WaitTime* which represents the maximum wait time for the reception of the replies expected. This time is proportional to the number *N* of messages awaited. The *CP* enters the role *receiver*, thus awaiting for water level information from the reporter peers. The LCC specification for this role comprises two main parts: one models the reception of the message *water_level* and the other shapes the time elapsing. The information embedded in the *water_level* message are: (1) an identification of the reporter *ReporterID*; (2) the identification of the location *Node*; (3) the time-step *Timestep* representing when the information was requested and, most important, (4) the value of the water level *WaterLevel* registered by the reporter at the location.

After having received the message, the *CP* stores the data just acquired (*update_flood_record* constraint) and waits for other similar messages from other reporters. If the wait time established by the *CP* expires and not all the reporters have replied, the *CP* terminates the interaction thus missing some water level data.

```

r(querier,initial)
r(sender,auxiliary)
r(receiver,auxiliary)
r(reporter,necessary,1,200)

a(querier,Q)::

null <- get_peer_name(PeerName) and getTimestep(Timestep) and
      getPeers("reporter",SPL) then

a(querier1(PeerName,Timestep,SPL),Q) then

null <- size(SPL,N) and getWaitTime(N,WaitTime) then

a(receiver(WaitTime,N),Q) then

null <- close_connection(Timestep)

```



```
a(querier1(PeerName, Timestep, SPL), Q) ::  
  
  null <- SPL=[]  
  
  or  
  
  (  
    null <- SPL=[H|T] then  
    request_flood_status(PeerName, Timestep) => a(reporter, H) then  
    a(querier1(PeerName, Timestep, T), Q)  
  )  
  
a(receiver(WaitTime, N), Q) ::  
  
  null <- equalZero(N)  
  
  or  
  
  (  
    water_level(ReporterID, Node, Timestep, WaterLevel) <= a(reporter, R) then  
    null <- update_flood_record(Node, WaterLevel, ReporterID, Timestep) and  
    dec(N, NewN) then  
    a(receiver(WaitTime, NewN), Q)  
  )  
  
  or //handle the wait-time elapsing  
  
  (  
    null <- equalZero(WaitTime)  
  
    or  
  
    (  
      null <- sleep(1000) and dec(WaitTime, NewWaitTime) then  
      a(receiver(NewWaitTime, N), Q)  
    )  
  )  
)
```

The *reporter* role is very straightforward: after having received the message *request_flood_status*, the reporter peer retrieves the water level sensed (*retrieve_flood_level* constraint). Notice that the value of the water level registered by a reporter may not correspond to the real value (e.g., the reporter peer is an untrustworthy peer). Once the water level is retrieved, the reporter peer sends the message *water_level* back to the requester.

```
a(reporter,R) ::  
  
request_flood_status(PeerName,Timestep) <= a(querier1,Q) then  
  water_level(ReporterID,Node,Timestep,WaterLevel) => a(receiver, Q)  
  <- retrieve_flood_level(ReporterID,Node,PeerName,Timestep,WaterLevel)
```

Appendix B

Simulator Interaction Models

In what follows, we describe the simulation cycles and the interaction models used by the simulator peers.

B.1 `Simulation_Cycles.lcc`

This is the main interaction model enacted by the controller module of the e-Response simulator. This interaction realizes the cycling needed to evolve the simulation. The only participant of this interaction is the controller itself. It plays the main role *simulator* (see LCC code below).

When the *simulator* role is entered, some parameters are instantiated, the connections with the flood sub-simulator and the physical peers are established and the simulation cycling is initiated. The parameters are:

- *MaxTimeStep*: total number of simulation cycles;
- *SimSleepTime*: amount of time the simulator stays idle after its main operations of gathering and sending info;
- *Timeout*: time awaited for peer connections, expressed in seconds. When this time expires the simulator starts the simulation cycles;

- *NoExpConnections*: number of expected peer connections. This number depends on the experiment and is used to compute the *Timeout*;
- *NoPeerConnected*: number of peers effectively connected. Its value is initiated to zero.

Once the above parameters are set, the controller attempts to connect with the available disaster sub-simulators by solving the constraint *connectWithSubSimulators*. Such constraint actually enacts the interaction model “Flood SubSimulator_Connection” described in the next section. After the termination of this interaction, the controller knows which sub-simulator is properly connected to it. If the sub-simulator connection fails or there are no sub-simulators available, the “Simulation_Cycles” interaction still goes on with the result of not simulating any disaster evolution. Once the sub-simulators are connected, the controller jump to the *connections_waiter* role during which the peer connections are awaited and counted¹. At this point, when both sub-simulators and peers are connected to the simulator, the cycling is initiated (*init_simulation_cycles* constraint), the first time-step is acquired through the *getCurrentTimestep* constraint and the info concerning the joined peers are prepared to be sent to the visualiser (*init_visualiser* constraint). The controller then enters the role *info_handler* which represents the core part of the simulation and, finally, terminates the interaction by solving the constraint *close_simulation*. Such constraint is used to close database connections, delete temporary files, etc.

```
r(simulator, initial)
r(connections_waiter, auxiliary)
r(info_handler, auxiliary)
```

¹Notice that the peer connections are actually established by means of the “Peer_Connection” interaction model which runs in parallel with the described interaction and is initiated by a peer willing to connect.

```

a(simulator, SIM) ::

null <- getSimulationCycles(MaxTimeStep) and
        getSimSleepTime(SimSleepTime) and
        getPeerConnectionParams(Timeout, NoExpConnections, NoPeerConnected) and
        connectWithSubSimulators(MaxTimeStep) then

a(connections_waiter(Timeout, NoExpConnections, NoPeerConnected), SIM) then

null <- init_simulation_cycles(MaxTimeStep) and
        getCurrentTimestep(CurrentTimestep) and
        init_visualiser(CurrentTimestep) then

a(info_handler(CurrentTimestep, MaxTimeStep, SimSleepTime), SIM) then
null <- close_simulation(MaxTimeStep)

```

As anticipated before, the role *connections_waiter* is used to await for a number of peer connections. When this role is entered, the parameters *Timeout*, *NoExpConnections* and *NoPeerConnected* are specified. Every second, the simulator retrieves the number *NewNoPeerConnected* of peers connected so far (*getNumConnectedPeers* role) and updates the timeout *NewTimeout*; it then recurses again to this role by passing the updated parameters. The role ends when either the timeout has elapsed or the number of peer connected is equal to the number *NoExpConnections* of expected connections. Notice that, if the role is ended because of a timeout, the peers actually connected are less than the ones expected and the interaction still continues. However, this fact does not prevent a peer to join the simulation after this phase.

```

a(connections_waiter(Timeout, NoExpConnections, NoPeerConnected), SIM) ::

null <- equal(NoPeerConnected, NoExpConnections)

or //This is to simulate the time elapsing

( null <- equalZero(Timeout)
  or
  (
    null <- sleep(1000) and dec(Timeout, NewTimeout) then
    null <- getNumbConnectedPeers(NewNoPeerConnected) then
    a(connections_waiter(NewTimeout, NoExpConnections,
                        NewNoPeerConnected), SIM)
  )
)

```

)
)

The role *info_handler* constitutes the kernel of this interaction model and dictates the sequence of the two main operations of the controller. These operations are the following:

- *Gathering*: the controller receives information about the changes that happened to the world: (a) it receives the flood changes from flood sub-simulator and (b) it receives other changes from the peers in the peer network that caused these changes (and verifies their validity);
- *Informing*: the controller sends information about the changes that happened in the world: (a) it sends changes (called sensory-info) that occurred in a peers vicinity to each peer in the peer network and (b) it sends a list of all the changes to the simulator’s visualiser.

The gathering operation is realized by the *gather_info* constraint. In such constraint, the flood sub-simulator connection state is first retrieved and then, if the flood sub-simulator is connected, the interaction model “Flood” is enacted in order to get the flood changes from the sub-simulator. The constraint ends by making the controller idle for an amount of time equals to *SimSleepTime*.

The informing operation is realized by the constraints *send_info* and *inform_visualiser*. In the *send_info* constraint, the interaction model “Sensory_info” is enacted in order to send contextual info to all connected peers. When this interaction terminates, the controller stays idle again for some times and the time-step counter is incremented. The *inform_visualiser* constraint is then solved to compute the changes occurred during the current time-step *CurrentTimestep*. For example, at time-step 2, such changes can assume the form:

```
updates (2, [[at (Tom,peer, [11.1207037, 46.0587387])],  
            [at (reporter3,reporter, [11.1116, 46.0968, 0.0])]]) .
```

The above format can be read as follows: at time-step 2, the fire-fighter Tom, which is a peer, is located at the geographical coordinates (11.1207037, 46.0587387); the reporter named “reporter2” is located at the geographical coordinates (11.1116, 46.0968) and its status set to 0 indicates that it is available to provide information on the water level present in its current location.

After the changes pertaining the current time-step are computed according to the above format, the controller enacts the interaction model “Visualiser” so to send the updates to the simulator’s visualiser. Once the *inform_visualiser* constraint is completed, the new time-step *NewTimestep* is retrieved (*getCurrentTimestep* constraint) and the controller recursively jumps back in the *info_handler* role to start a new simulation cycle. The role, and hence the whole interaction, terminates only when the new time-step is greater than the maximum number of cycles *MaxCycles* foreseen for the simulation.

```
a(info_handler(CurrentTimestep, MaxTimeStep, SimSleepTime), SIM) ::  
  
  null <- greater(CurrentTimestep, MaxTimeStep)  
  
  or  
  
  (  
    null <- gather_info(SimSleepTime) and  
           send_info(SimSleepTime) and  
           inform_visualiser(CurrentTimestep) and  
           getCurrentTimestep(NewTimestep) then  
  
    a(info_handler(NewTimestep, MaxTimeStep, SimSleepTime), SIM)  
  )
```

B.2 Flood Sub-Simulator_Connection.lcc

This interaction model is played by the controller and the flood sub-simulator; it is used to get the topology and connect the sub-simulator to the controller. The topology defines the flooding areas, the geographical coordinates of the locations (included strategic locations such as meeting points, refuge centers, etc.) and their connections. This interaction model can be extended so to connect the controller to many disaster sub-simulators.

As can be seen below, the peer playing the *controller* role sends a topology's URI to the peer playing the *sub-simulator* role with the aim that both peers, during the current simulation, use the same topology of the world. Note that the flood sub-simulator works in parallel with the controller. Since the flood sub-simulator does not have neither data nor equations to simulate flood evolution in all the world but only in Trento town, after downloading the topology it first verifies if in its local database there is the data that should be used to simulate the flood evolution in the region received and then it does some other initializations, like joining the selected data using a geospatial query.

The second aim of this interaction model is to store the flood sub-simulator connection state in the local knowledge of the controller . The connection state of the sub-simulator is set to “successfully connected” only if the sub-simulator downloads the topology file without any failures. This state is then verified in the constraint *gather_info* of the previous interaction model. At each time-step, if this state is successfully verified, the flood interaction model (see section B.4) is then enacted.

```
r(controller,initial)
r(sub_simulator,necessary)

a(controller,C) ::
```



```
initial_topology_source(URI) => a(sub_simulator,SS)
  <- getInitialTopology(URI) then
  (
    (got_topology(URI) <= a(sub_simulator,SS) then
      null <- setFloodSubSimConnection("true"))

    or

    (connection_failure(URI) <= a(sub_simulator,SS) then
      null <- setFloodSubSimConnection("false"))
  )

a(sub_simulator,SS) ::

initial_topology_source(URI) <= a(controller,C) then
  (
    got_topology(URI) => a(controller,C) <- getTopology(URI)

    or

    connection_failure(URI) => a(controller,C)
  )
```

B.3 Peer_Connection.lcc

This interaction model is used to connect a physical peer to the simulator and is initiated by the peer willing to join the simulation. The main roles are *connecting_peer* and *registrar* which are played by a joining peer and the controller respectively. The controller subscribes to this interaction with the option of running in parallel many interactions of this type. In this way, an unfixed number of peers may connect to the simulator. This interaction remains active till the end of the simulation.

When the peer enters the *connecting_peer* role (see LCC code below), it first retrieves its characterizing parameters (e.g., *PeerName*, *PeerType*, *Location*) and then sends the message *exist* to the controller. The message *connected* is thus received as reply from the controller. The following parameters are specified in the message:

- *RegisteredName*: the name registered by the controller to identify the connecting peer;
- *TS*: the time-step at which the connection takes place;
- *MaxTimestep*: the duration (in time-steps) of the simulation;
- *SimSleepTime*: the time (expressed in seconds) used to estimate how long the connecting peer should wait for an incoming message;
- *WLThr*: the water level threshold above which a node (a location in the topology) is blocked.

The above parameters, when received, are stored in the peer local knowledge through the constraint *updateSimParameters*. After this operation, the peer enters the *connected_peer* role.

```

r(connecting_peer, initial)
r(connected_peer, auxiliary)
r(interrupter, auxiliary)
r(registrar, necessary)
r(registrar2, auxiliary)

a(connecting_peer, Id) ::

exists(PeerName, PeerType, Location) => a(registrar, S)
<- get_peer_name(PeerName) and
   connect(PeerName, PeerType, Location) then

connected(RegisteredName, TS, MaxTimestep, SimSleepTime, WLThr)
  <= a(registrar, S) then

null <- updateSimParameters(RegisteredName, TS, MaxTimestep,
                             SimSleepTime, WLThr) then

a(connected_peer(MaxTimestep, PeerName), Id)

```

For all the duration of the simulation, the peer maintains the *connected_peer* role (see LCC code below). This role starts by retrieving the current time-step *Timestep*. This time-step is checked against the maximum

number of time-steps (*MaxTimestep*) foreseen by the simulation. If the current time-step overcomes the *MaxTimestep*, the simulation terminated, the peer disconnects and the *connected_peer* role can be stopped. In the other case, the simulation is evolving and the peer may decide (*disconnect* constraint) to exit from it, or to pause it (through *start_pause_simulation* constraint) and resume it again. When the peer wants to temporarily disconnect, the message *await_decision* is sent to the controller and the role *interrupter* is taken. Once in this role, the peer continuously recurses till the *stop_pause_simulation* constraint becomes true; when this happens the message *decision_made* is sent to the controller, meaning that the peer intends to resume the simulation. The peer goes therefore back to the *connected_peer* role.

```
a(connected_peer(MaxTimestep,PeerName),Id) ::  
  
  null <- getTimestep(Timestep) then  
  
  (  
    null <- greaterOrEqual(Timestep,MaxTimestep) and disconnect() then  
  )  
  
  or  
  
  (  
    exit(PeerName) => a(registrar2,S) <- disconnect()  
  )  
  
  or  
  
  ( await_decision(PeerName) => a(registrar2,S)  
    <- start_pause_simulation(T) then  
    a(interrupter,Id) then  
    a(connected_peer(MaxTimestep,PeerName),Id)  
  )  
  
a(interrupter,Id) ::  
  decision_made => a(registrar2,S) <- stop_pause_simulation()  
  or  
  a(interrupter,Id)
```

The *registrar* role is the main role taken by the controller (see LCC code below). It handles the first phase of the peer connection, that is, the reception of the *exist* message from the connecting peer. First, it retrieves the maximum number of time-steps *MaxTimesteps*², then it waits for an incoming *exist* message. Once such message is received, the controller registers the peer identity with a name *RegisteredName* (*register* constraint) and adds it to the simulation (*add_peer_to_sim* constraint). In this way, the peer location is also registered and the current time-step *Time*, representing the registration time, is obtained. Also, the parameters *SimSleepTime* and *WLThr* are retrieved through the constraints *getSimSleepTime* and *getWLThreshold* respectively. If the simulation is running, these parameters are then sent back to the connecting peer via the *connected* message. The controller thus takes the role *registrar2* till the end of the simulation.

```

a(registrar, S) ::

null <- getMaxTimesteps(MaxTimesteps) then

(
exists(PeerName, PeerType, Location) <= a(connecting_peer, Id) then
null <- register(Id, PeerType, PeerName, RegisteredName) and
add_peer_to_sim(PeerName, PeerType, Location, Time) and
getSimSleepTime(SimSleepTime) and getWLThreshold(WLThr) and
lessOrEqual(Time, MaxTimesteps) then

connected(RegisteredName, Time, MaxTimesteps, SimSleepTime, WLThr)
=> a(connecting_peer, Id) then

a(registrar2(MaxTimesteps, PeerName), S)
)

```

The *registrar2* role is entered by specifying the two parameters *MaxTimesteps* and *PeerName* (see LCC code below). The current time-step *Time-step* is first obtained through the constraint *getTimeStep*. Then, the controller can receive two types of messages from the connected peer: *exit*

²Notice that this parameter is set in the “Simulation_cycles” interaction model which started first and is running in parallel.

and *await_decision*. If the first message is received, the controller performs the constraint *remove_peer_from_sim* to definitively disconnect the peer from the current simulation and ends the *registrar2* role of this running instance of interaction. If the second message is received, the controller solves the constraint *await_decision* which temporarily exclude the peer from the simulation till the message *decision_made* is received from the peer. The peer is therefore resumed and the controller recurses again to this role. The role finally terminates when the *MaxTimesteps* are reached.

```
a(registrar2(MaxTimesteps,PeerName),S) ::  
  
  null <- getTimestep(Timestep) then  
  
  (  
    null <- greaterOrEqual(Timestep,MaxTimesteps)  
  )  
  
  or  
  
  (  
    exit(PeerName) <= a(connected_peer,Id) then  
    null <- remove_peer_from_sim(PeerName)  
  )  
  
  or  
  
  (  
    await_decision(PeerName) <= a(connected_peer,Id) then  
    null <- await_decision(PeerName) then  
    decision_made(Empty) <= a(interrupter,Id) then  
    null <- end_await_decision(PeerName) then  
    a(registrar2(MaxTimesteps,PeerName),S)  
  )
```

B.4 Flood.lcc

This interaction model is used by the controller at every time-step, in order to get from the flood simulator the changes of the flood level registered at the nodes in the topology.

```

r(controller, initial)
r(flood_simulator, necessary)

a(controller,C) ::

null <- getTimeFlood(Time) then

(
  request_info(Time) => a(flood_simulator,FS) then
  flood_info(Changes) <= a(flood_simulator,FS) then
  null <- updateFloodChanges(Changes)
)

a(flood_simulator,FS) ::

request_info(Time) <= a(controller,C) then
  flood_info(Changes) => a(controller2,C) <- floodChanges(Time,Changes)

```

This interaction model is enacted by the constraint *gather_info* in the “Simulation Cycles” IM of section B.1, which manages cycles and therefore also time-step increments.

The starting role of the “Flood” IM is the ‘controller’ role that is played by the controller peer. It first gets the current time-step and then it sends a message to the flood sub-simulator requesting water level changes at the current time. After receiving an answer with flooding changes, it updates its local knowledge of the world with the acquired information. The update is made within the core constraint of this role, i.e., the *update-FloodChanges(Changes)* constraint. The Java method implementing such constraint invokes a Prolog query³.

The other role, *flood_simulator* is played by the flood sub-simulator peer. In this role, the core constraint is *floodChanges(Time,Changes)*. Here the flooding law (8.1) is implemented. For each node that has been affected by some changes in flood status, it sends a message to the controller with flooding changes in the form:

³Some basic code of the controller peer is left in Prolog

```
nodeFloodLevel(nodeid, level)
```

where `nodeid` is the identifier of the node received in the topology file at initialization time and `level` is a real number in $[0, 3]$ range indicating the level of water in meters. The following conditions are assumed depending on the water level values:

- `level < 0.5`: no critical water
- `level > 0.5`: stretch of road blocked
- `level >= 2`: person dead

The following is an example of the content of the *Changes* argument in the *floodChanges(Time, Changes)* constraint:

```
[nodeFloodLevel(23, 0.3), nodeFloodLevel(45, 1.2), nodeFloodLevel(66, 2.2)]
```

B.5 Sensory_Info.lcc

This interaction model is initiated by the controller at every time-step. In particular, it is enacted in the constraint *send_info*, within the interaction model “Simulation-cycle” described in B.1. It is used to send contextual information (sensory-info) to all connected peers. Such information depend on the recipient peer and are represented by the following parameters:

- *PeerName*: the name of the recipient peer;
- *Timestep*: the time-step referred by the sensory-info;
- *Location*: the current location of the peer;
- *Flood*: the water level registered at the location where the peer is;

- *SimName*: the name identifying the simulator;
- *NeighPeers*⁴.: a list of neighbors peers, that is, peer located in the vicinity of the recipient peer *PeerName*.

The interaction model comprises two main roles, *sensory_info_sender* and *connected_peer*, which are taken by the controller and a connected peer respectively (see LCC code below).

The controller starts the interaction by entering the *sensory_info_sender* role where the parameters *SimName* and *Timestep* are retrieved and the peer list *PL* of all connected peers is obtained. The controller then jumps to the role *sensory_info_sender1* to actually compute and send the sensory-info to each peer in the list. The sensory-info are computed by solving the constraint *send_update_info* that takes as input the peer identifier *H*, which is assigned by the kernel, in order to extract the registered name of the peer and hence its real name *PeerName*. Based on the *PeerName*, the parameters *Location* and *Flood* are then obtained. The message *sensory_info* is thus sent to the current peer and the role recurses to handle the sensory-info of the subsequent peer.

Each connected peer plays the *connected_peer* role. The peer simply awaits for the *sensory_info* incoming message and then performs an update of the parameters received (*Timestep*, *Location*, *Flood*, *NeighPeers*) by means of the constraint *update_info*. After the update, the role is ended. The recursion is not needed since a peer connected to the simulation subscribes to this interaction with an acceptance policy of “all”, meaning that the peer can execute more than one interaction of this type.

```
r(sensory_info_sender,initial)
r(sensory_info_sender1,auxiliary)
r(connected_peer,necessary,1,75)
```

⁴Though the simulation is predisposed to handle this parameter, its computation is still missing and, therefore, this parameter is always a blank list


```
a(sensory_info_sender, S) ::

  null <- get_peer_name(SimName) and
          getControllerTimestep(Timestep) and
          getPeers(``connected_peer'', PL) then

  a(sensory_info_sender1(SimName, Timestep, PL), S)

a(sensory_info_sender1(SimName, Timestep, PL), S) ::

  null <- PL=[]

  or

  (
    null <- PL=[H|T] then
    (
      sensory_info(Timestep, SimName, PeerName, Location, Flood, NeighPeers)
      => a(connected_peer, H)
      <- send_update_info(Timestep, H, PeerName, Location, Flood, NeighPeers)
    ) then

    a(sensory_info_sender1(SimName, Timestep, T), S)
  )

a(connected_peer, Id) ::

sensory_info(Timestep, SimName, PeerName, Location, Flood, NeighPeers)
<= a(sensory_info_sender1, S) then

  null <- update_info(Timestep, SimName, PeerName, Location, Flood, NeighPeers)
```

B.6 Visualiser.lcc

This interaction model is used to let the controller inform the visualiser of all the changes that have occurred in the world at every time-step. It is enacted in the constraint *inform_visualiser*, within the interaction model “Simulation-cycle” described in B.1. This ensures changes are sent out only once every time-step.

The controller plays the *controller* role (see LCC code below). After

having retrieved the current time-step *CurrTime* and get the previous time-step *PrecTime*, a check is done on the latter parameter to find out if the current time-step is the first time-step of the simulation. If so, initial information are retrieved and then sent to the visualiser. Such information regards: (i) the water level threshold which establishes the maximum water level above which a node is blocked; (ii) the peers who currently joined the simulation; (iii) the initial positions of all connected peers, the location of the reporters and their initial status. Notice that these information are got by solving the constraints *getThrInfo*, *getJoinInfo* and *getAtInfo* respectively and are sent via the *initInfo* message only once⁵. For time-steps greater than 1, a unique constraint (*getAllChanges*) is solved which retrieves information of type (iii). The information thus retrieved, which are contained in the parameter *AllChanges*, are then sent to the visualiser via the *changes* message. The parameter *CurrTime* is also incorporated in the message. The parameter *AllChanges* looks like the following, whose meaning is explained in section B.1:

```
updates(2, [[at (Tom,peer, [11.1207037, 46.0587387])],
            [at (reporter3,reporter, [11.1116, 46.0968, 0.0])]]) .
```

```
r(controller,initial)
r(visualiser,necessary)

a(controller,C) ::

null <- getCurrentTimestep(CurrTime) and
assign(CurrTime,CurrTime1) and
dec(CurrTime1,PrecTime) then
(
initInfo(CurrTime,ThrInfo,JoinInfo,AtInfo) => a(visualiser,V)
null <- equalZero(PrecTime) and
getThrInfo(ThrInfo) and
getJoinInfo(JoinInfo) and
```

⁵The constraints *getThrInfo*, *getJoinInfo* and *getAtInfo* only retrieve the information which are actually computed within the constraint *inform_visualiser* of the “Simulation_cycle” interaction model described in section B.1.

```
        getAtInfo(CurrTime,AtInfo)
    )
or
(
  changes(CurrTime,AllChanges) => a(visualiser,V)
  null <- getAllChanges(CurrTime,AllChanges)
)
```

The visualiser plays the *visualiser* role. By receiving the *initInfo* message, it starts its GUI with the parameter acquired (*start_visualiser*). If a *changes* message is received instead, it updates its history according to the new information (constraint *updateChanges*). The update results in a change on the GUI.

```
a(visualiser,V) ::
(
  initInfo(ThrInfo,JoinInfo,AtInfo) <= a(controller,C) then
  null <- start_visualiser(ThrInfo,JoinInfo,AtInfo)
)
or
(
  changes(Timestep,AllChanges) <= a(controller,C) then
  null <- updateChanges(Timestep,AllChanges)
)
```

B.7 Perform_Action.lcc

This interaction model is used to let the connected physical peers inform the controller of the physical actions they are performing. As mentioned earlier, peers should inform the controller of all their physical actions since these would result in changes in the physical world. Furthermore, it is the controller that would confirm whether an action is currently possible or not. This interaction model is executed every time a connected physical

peer needs to perform an action. In particular, its enaction takes place in the constraint *try_move_action* of the “Evacuation” interaction model described in the next section. Although the action in question is always a “move” action, this interaction model is designed to be usable for any kind of action.

The connected physical peer initiates the interaction by entering the *action_performer* role (see LCC code below). Here, the parameters *RegisteredName* and *Action* are retrieved, by means of *get_registered_name* and *get_peer_action* constraints, in order to send the *action* message. The parameter *Action* specifies the action the peer attempts to perform; in our simulation⁶, it is expressed by the string “*move(N1,N2, Vehicle)*”, where *N1*, *N2* and *Vehicle* identify respectively the initial position, the final destination and the mean of transport used to move. After having sent the *action* message, the moving peer receives the *action_state* message from the controller. Such message contains the parameter *ActionState* which specifies whether the action has been performed or stopped by the simulator. In any case, the value of the parameter is stored in the local knowledge of the connected peer through the *set_action_state* constraint. This interaction does not tell anything about the future actions the peer will take depending on the result received. In our simulation, this kind of issues are dealt with in the peer’s OKCs rather than in the LCC code. This guarantees a major flexibility in the interaction model which can thus be used in more general contexts.

```
r(action_performer, initial)
r(simulator, necessary, 1)

a(action_performer, P) ::
(
  action(RegisteredName, Action) => a(simulator, S)
  <- connected() and
```

⁶Though generic, the actions currently performed are the “move” actions only.

```

    get_registered_name(RegisteredName) and
    get_peer_action(Action) then

    action_state(ActionState) <= a(simulator,S) then

        null <- set_action_state(ActionState)
    )

```

The simulator’s controller plays the *simulator* role (see LCC code below). Its aim is to tell the connected peer whether it can perform the action or not. The controller subscribes to this interaction at the very beginning of the simulation with an acceptance policy of “all”. This guarantees that the controller can serve multiple requests from the connected peers. The controller first checks whether the simulation has terminated or not. If yes, the role is ended otherwise the *action* message is received. The constraint *update_action_results* is thus solved in order to evaluate the possibility of executing the action. In particular, being the action in question a “move” action, the controller checks the action feasibility by determining the flood level of the destination specified in the *Action* parameter; if the associated stretch of road is blocked, the controller set the parameter *ActionState* to “stopped”, this meaning that the peer cannot perform the action. On the contrary, if no risk is associated to the piece of road, the controller updates the position of the moving peer to the new location (the destination) and sets the value of *ActionState* to “performed”. After this process, the message *action_state* is finally sent to the connected peer.

```

a(simulator,S)::

    null <- getMaxTimestep(MaxTimestep) and getControllerTimestep(Timestep) then

    (
        null <- greaterOrEqual(Timestep,MaxTimestep)
    )

    or

    (

```

APPENDIX B. SIMULATOR INTERACTION MODELS

```
action(RegisteredName,Action) <= a(action_performer,P) then
  action_state(ActionState) => a(action_performer,P)
  <- update_action_results(RegisteredName,Action,ActionState)
)
```