UNIVERSITY
OF TRENTO

DIPARTIMENTO DI INGEGNERIA E SCIENZA DELL'INFORMAZIONE

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.disi.unitn.it

D6.8: SUMMATIVE REPORT ON THE USE OF OPENKNOLEDGE FRAMEWORK IN E-RESPONSE: INTEGRATION AND EVALUATION RESULTS

Gaia Trecarichi, Veronica Rizzi, Lorenzino Vaccari and Maurizio Marchese

February 2009

Technical Report # DISI-09-016

OpenKnowledge

FP6-027253

D6.8: Summative report on the use of OpenKnoledge framework in e-Response: integration and evaluation results

Gaia Trecarichi[1], Veronica Rizzi[1], Lorenzino Vaccari[1], Maurizio Marchese[1]

[1] University of Trento
{gtrecari;vrizzi;vaccari;marchese}@disi.unitn.it

Report Version: final
Report Preparation Date: 31/12/08
Classification: deliverable 6.8
Contract Start Date: 1.1.2006 Duration: 36 months
Project Co-ordinator: University of Edinburgh (David Robertson)
Partners: IIIA(CSIC) Barcelona
Vrije Universiteit Amsterdam
University of Edinburgh
KMI, Open University
University of Southampton
University of Trento

Abstract

This deliverable aims at investigating the capability of the OpenKnowledge framework to support centralised as well as decentralised architectures for information gathering in emergency response management. For this purpose, we developed an agent-based e-Response simulation environment fully integrated with the OpenKnowledge infrastructure and through which existing emergency plans are modelled and simulated. Preliminary results show (1) the overall scalability of the OpenKnowledge kernel to realistic use cases; (2) the capability of the OpenKnowledge framework in supporting the two afore-mentioned architectures and, under ideal assumptions, a comparable performance in both cases.

1 Introduction

All phases of emergency response management - that in the following we will reference as emergency response (e-Response) activities - depend on data from a variety of sources. Moreover, during an emergency it is critical to have the right data, at the right time, displayed logically and contextually, to respond and take the appropriate actions. At present, most of the information management infrastructures required for dealing with emergencies are based on centralised architectures that (i) are specifically designed prior to the emergency, (ii) gather centrally the available information, (iii) distribute it upon request to the appropriate agents (e.g., emergency personnel, doctors, citizens). While centralised infrastructures provide a number of significant advantages (in terms of quality control, reliability, trustworthiness, sustainability, etc.), they also present some well-known intrinsic problems (e.g., physical and conceptual bottlenecks, communication channel overloads, single point of failure). All these issues are taken into account in the design and deployment of current mission-critical centralised systems. However, information sharing breakdowns are still possible and have occurred also in recent emergency events, such as the catastrophic passage of Hurricane Katrina in New Orleans in 2005[1]. Alternative data management (both for gathering and providing information) infrastructures are currently being explored, studied and analyzed ([1], [2], [3]) in order to support data sharing also in the absence of a centralised infrastructure. In this study, we explore the flexibility and adaptability of the OpenKnowledge framework in the context of an e-Response scenario. This framework provides a distributed infrastructure, that enable peers to find and coordinate with each other by publishing, discovering and executing interaction models, i.e. multi party conversational protocols written in the Lightweight Coordination Calculus (LCC)[4]; the key novelty of the approach is that no a-priori agreement or knowledge of the conversation partners is needed to have meaningful interactions. In this

[1] http://en.wikipedia.org/wiki/Effect_of_Hurricane_Katrina_on_New_Orleans

work, the proposed OpenKnowledge (OK) infrastructure is used to explore its capability to support both centralised and decentralised architectures for information gathering in open environments. For this purpose, we built a simulation-based test-bed fully integrated with the OK platform. The final goal of such virtual environment is to evaluate this framework in the e-Response domain. In particular, we implemented an e-Response simulation environment through which existing emergency plans based on real-data are modelled and simulated. Moreover, a suite of experiments has been designed and run to evaluate the performance of the OK e-Response system under specific assumptions. Preliminary results show the system's capability of supporting the two afore-mentioned architectures and a comparable performance in both cases. To summarize, the main contributions of the present deliverable are:

- The full use and testing of the current release of the OpenKnowledge infrastructure in a realistic and demanding use case;
- The provision of an agent-based simulation environment in which to evaluate interaction models, coordination tasks and diverse emergency information-gathering models;
- A preliminary analysis and comparison between the effectiveness of the OpenKnowledge infrastructure in centralised (hierarchical) and decentralised (p2p) information gathering in e-Response management activities.

The idea to explore and test the effectiveness of different data management architectures in "real-world" e-Response setting is not new. It is recognized [5] that realistic computer simulations can be a valuable tool to investigate innovative solutions, such as new collaborative information systems, new cooperation configurations and communication devices. In fact, several multi agent-based simulation applications have been developed in diverse domains ([6], [5], [7], [8], [9]). Related research projects are either specifically devised for the emergency management area or focused more on the architectural aspect. In particular, CASCOM[2], WORKPAD[3], EGERIS[4], EUROPCOM[5], POMPEI[6], POPEYE[7] and WIN[8] are among such projects. For example, 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 [10]. Also, in the FireGrid project [11], a software architecture to help fire-fighters in e-Response events

[2]http://www.ist-cascom.org
[3]http://www.workpad-project.eu/description.htm
[4]http://www.egeris.org
[5]http://www.ist-europcom.org
[6]http://www.pompei-eu.com
[7]http://www.ist-popeye.org
[8]http://www.win-eu.org

has been built. They realized an integrated system where real-time sensor data are processed using sophisticated models, running on High Performance Computing (HPC) resources accessed via a Grid interface, and finally presented to humans using a command-and-control multi-agent system. In this case, a mechanism based on the OpenKnowledge approach would allow each agent to execute, and eventually modify, the workflow, thanks to the sharing of the multi-agent protocol.

In what follows, we first introduce the e-Response test-bed (Section 2). We then present, in Section 3, the e-Response case study used to experiment the OK framework. Next, in Section 4, we describe the e-Response simulation environment architecture and, in Section 5, we present the experimental summative experiment designed for the evaluation; preliminary results of centralised vs. decentralised information management architectures are also discussed. In Section 6, we draw our conclusion and future work. Section 7 contains akcnowledgments. Finally, in Appendixes A-1 and A-2, we provide a technical documentation of all the interaction models developed for the various experiments.

2 The e-Response Test-bed

The developed e-Response test-bed consists of an agent-based e-Response simulation environment fully integrated with the OpenKnowledge infrastructure and through which existing emergency plans are modelled and simulated. In particular, with such test-bed we:

- Simulate significant e-Response use-cases using the OK infrastructure;
- Investigate how the OK framework is capable of supporting emergency activities coordination (centralised vs. decentralised information gathering);
- Test the robustness of the OK kernel by exploring two dimensions: (1) the number of peers involved in a simulated coordination task and (2) the number of interaction models;

The e-Response test-bed is composed of the following main components which will be described through the rest of the deliverable:

1. A Simulator capable of modelling a flood event in Trentino, using real GIS/flood data (see section 4.2);
2. Suite of LCC interaction models (ca.13) supporting three different peer coordination strategies: baseline, centralised and decentralised coordination. These strategies will be discussed in section 3.2.2; full details on the interaction models are given in appendixes A-1 and A-2;

3. Suite of OpenKnowledge components (ca. 25) enabling emergency peer types. Such peer types are described in section 3;
4. Suite of Peers (ca. 300) modelling emergency agents;
5. Suite of experiments aimed at testing the OK kernel (more details in section 5);
6. ICT Infrastructure to support the experiments (DBs,servers,script); details on part of the infrastructure can be found in section 5.3.

3 The e-Response Case Study

In this section, we describe the case study where the OpenKnowledge framework has been applied: an emergency response scenario. The nature of the specific e-Response domain is such that a structured coordination is necessary in order to prevent chaotic and uncontrolled conditions. Nevertheless, taking into account flexibility is fundamental to handle unexpected situations (e.g., sudden road blockage, fast and unpredicted events, etc) which will most likely happen in emergency situations. While the general vision of interaction protocols accounts for the structured coordination requirement of the problem, the adoption of models specifically designed to explicit interactions in a p2p fashion and passed through an underling open infrastructure accounts for the support of flexibility and dynamicity.

We applied the OpenKnowledge framework in the case study of a flooding disaster in Trento (Italy). The work moved its steps 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 previous OpenKnowledge deliverables ([12],[13]). Figure 1 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 down part relates to the *evacuation* phase.

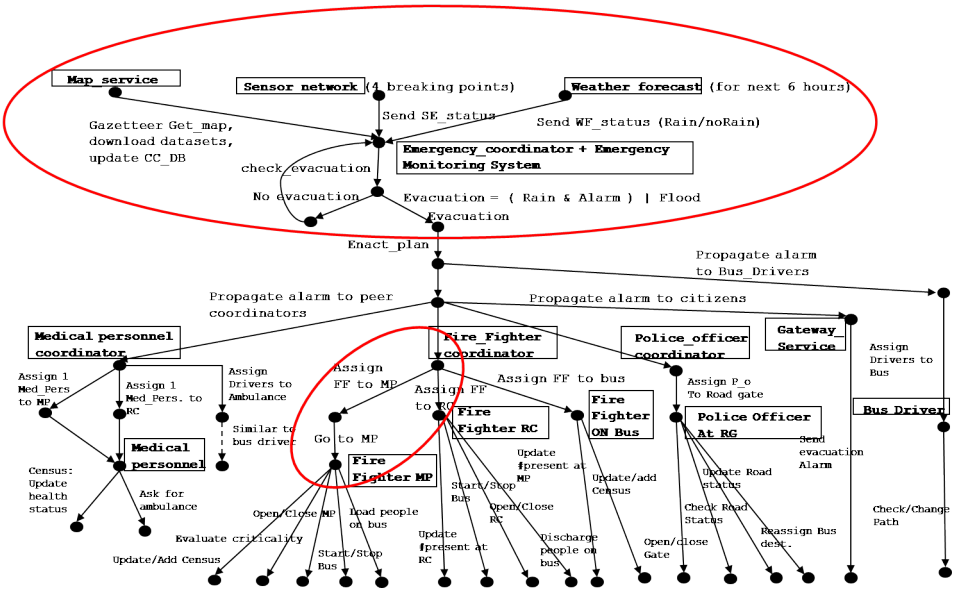

Figure 1: The overall e-Response use case

In the prealarm phase are mainly involved service peers which are, as has been previously said, peers providing all that information needed to enact the emergency plan or not. The pre-alarm phase thus involves mainly service peers which provide information useful for decision making. The pre-alarm phase eventually results in the evacuation phase. Such phase regards all the activities needed to move people to safe places. In such 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). Of course, the emergency peers are supported by service peers such as route services, sensors scattered across the emergency area, etc.

Figure 2 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 test-bed and will be illustrated in detail in the next section 4.

The figure also shows two different evacuation sub-scenarios: in both of them, a peer needs to get information on route's practicability but while in one case (area above the red line) such moving peer (MP) gets route information by asking the Civil Protection (CP), in the other one (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 centralised and decentralised strategies.

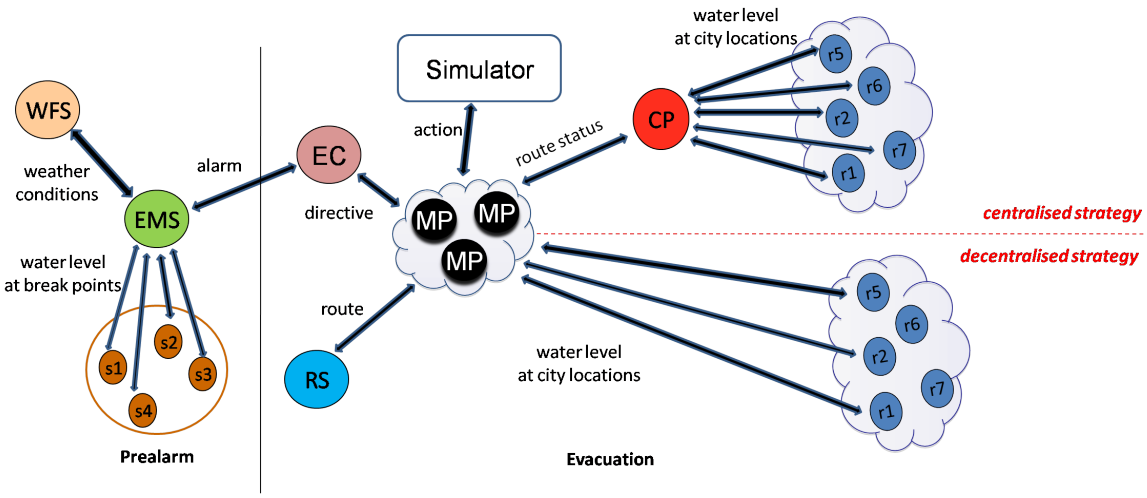

Figure 2: The implemented e-Response scenarios

3.1 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 in order to be able to take the proper actions.

3.1.1 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 chief;
- *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 autority 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.

3.1.2 Prealarm interactions

The main interactions pertaining the prealarm phase - and which are modeled in terms of LCC interaction models - are essentially three. A short description for each of them follows:

1. *Water level sensors monitoring*: a central monitoring system (EMS) requests continuously 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 information collection*: the emergency chief requests periodically a weather forecast (i.e., rain and temperature) in order to make previsions and therefore decisions on the actions to take.
3. *Alarm message generation*: when certain water thresholds and weather conditions are detected, an alarm is sent to the emergency chief.

Appendix A-1 contains a more detailed description and the LCC code relative to the above interactions.

3.2 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 some activities, i.e., choosing a path to follow (usually by asking a route service), checking if the path is practicable (usually by interacting with the Civilian Protection or with available reporters distributed in the area), proceeding along the path. The Civilian 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.

3.2.1 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)
- *Moving Peer (MP)*: it is a peer (e.g., an emergency subordinate as a fireman, a bus, a citizen) that needs to move to a specific location;
- *Route Service (RS)*: provides a route connecting two given locations; it can also provide a route that does not pass by a given set of undesired locations;
- *Civil Protection (CP)*: it is responsible for giving information on the blockage state of a given path;
- *Reporter (R)*: it 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.

3.2.2 Information gathering strategies

An important part of the evacuation scenario consists of checking whether a given route is practicable or not. However, in our simulation, we foresee also 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 useful information, constitutes the *baseline* scenario: a route is taken without checking a priori its conditions. The difference between this baseline scenario and the one developed in Deliverable 6.7 lies in the complete integration of the e-Response simulation environment with the OK kernel: all peers are now equipped with OKCs components and execute LCC interactions by exploiting the search-and-discover, the matching and the trust functionalities provided by the OK kernel.

Aside from focusing on porting the OK kernel into the simulation environment, the main effort of this year activity has been to design and implement experiments testing the capability of the OpenKnowledge platform to support two different information gathering strategies. These strategies relate to how a moving peer gather information on the route practicability, more specifically:

- ***Centralised strategy***: a moving peer obtains information on the blockage conditions of a given route after consultation with the Civil Protection *CP* (see Figure 2 - area above the red line);
- ***Decentralised 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 2 - area below the red line).

More details on how the moving peer reasons about the information thus gathered is given in section 4.1, where the peer-network component of the e-Response simulation system is described.

What we want to underline here is that the adoption of an information-gathering strategy (either centralized or decentralized) supports the peer in performing its task. Figure 3 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.

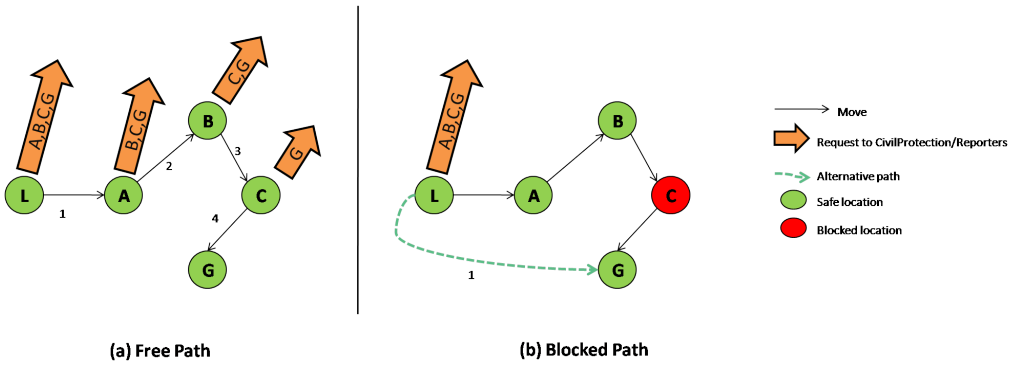

Figure 3: Evacuation phase: moving peer behaviour

3.2.3 Evacuation interactions

The main actions involved in the evacuation phase are shortly described below:

1. *Start 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 Civilian Protection needed to know the blockage state of a path;

4. *Gather real-time data from reporters*: a peer asks information about the water level to a group of reporters.

The above actions correspond respectively to the "Evacuation", "Find-Route", "Check-Route-State" and "Querier-Reporter" LCC interaction models. More information on the LCC-specifications used to describe the above scenarios and a full explanation of the LCC code can be found respectively in section 4.1 and Appendix A-2.

4 The e-Response System Architecture

To fully use and test the current release of the OpenKnowledge infrastructure in the realistic use case previously described, we built an e-Response simulation environment. The current simulation environment is based on the system presented in [14] and extends it both in a complete integration with the OpenKnowledge kernel and in the inclusion of a realistic flood-simulator. In particular, the following features can be found in this current version of our e-Response simulation system:

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 OpenKnowledge components: the LCC Interpreter, the Discovery Service, the Trust, Matching and GEA 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 current 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's types: the current simulation system extends the previous one in the number of peer types involved in the emergency scenario. New peer types such as Reporters, Civil Protection Unit, Weather Services, Emergency Monitoring Systems 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 current simulation the peer can adopt two different information-gathering strategies to ask for the route conditions.

The test-bed is used to evaluate interaction models, coordination tasks and the diverse emergency information-gathering models; through simulations, it is possible to estimate how the platform could perform in realistic emergency scenarios. The developed e-Response simulation system is used to: (1) model the behaviour of each peer involved in an e-response activity, (2) execute predefined interaction models within a p2p infrastructure and (3) visualize and analyze a simulated coordination task through a Graphical User Interface (GUI). The e-Response system is composed of two main components: the peer network and the e-Response simulator. Figure 4 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 greys arrows indicate interactions among network peers only. In the next three subsections, we illustrate the peer network, the e-Response simulator and the reuse of OK-components respectively.

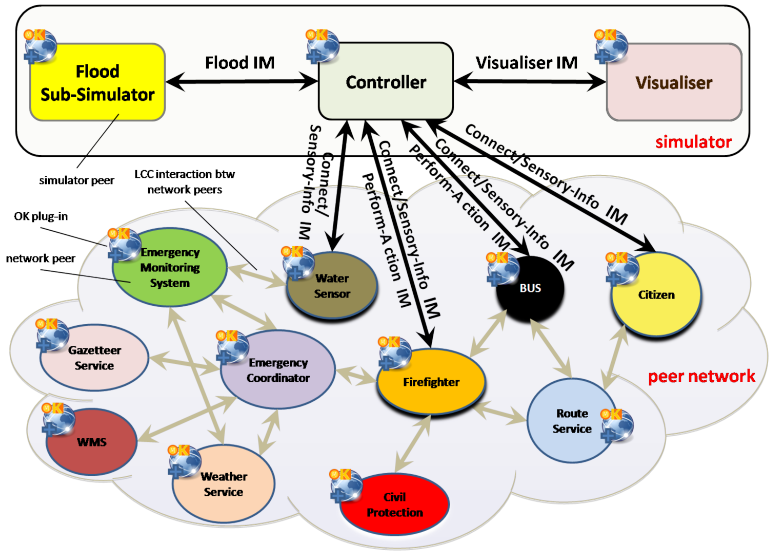

Figure 4: The e-Response system's architecture

4.1 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 some 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 4).

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 not shaded ellipses in Figure 4.

In what follows, we describe in more details those interactions between the network peers which regard the evacuation phase, that is, the phase which was simulated in order to test the OK infrastructure and compare the mentioned information gathering strategies. Figure 5 shows the architecture of the system where the main interactions between network peers are specified.

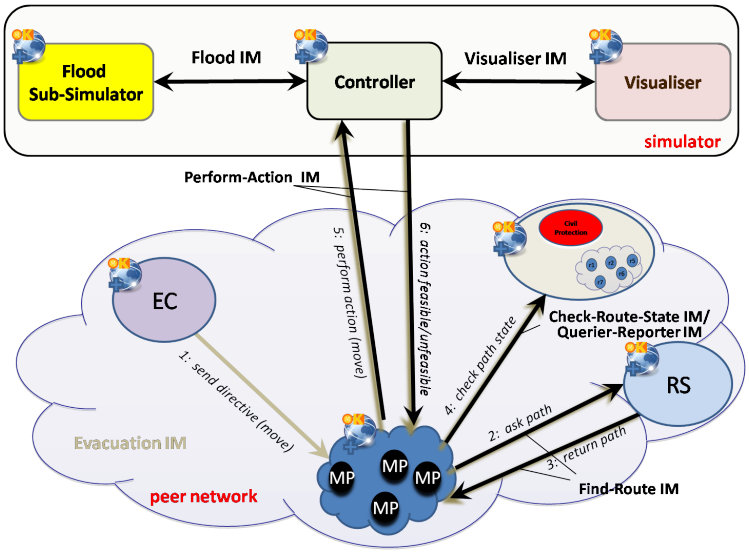

Figure 5: Evacuation phase: network peer's interactions

The whole 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* (centralised scenario) or the reporters r (decentralised 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 coded in terms of the "Evacuation" interaction model[9] which represents the "main" one in that it captures the whole evacuation scenario. In Figure 5, 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 interaction models. This is a key functionality of the OK platform, since it allows to write simple, modular and reusable LCC specifications. We tell something more on this later in this section.

Here, the "Evacuation" IM is described in its main parts, however, a detailed description can be found in appendix A-2.2.1. It simulates the evacuation phase and can be used in all those situations where an emergency chief sends the directive of reaching specific locations to its subordinates. In short, an emergency subordinate ES[10] receives an alert message from the chief and resolves some constraints in order to set the goal to be achieved (reach the goal destination G) and 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 6-7 show LCC code snippets for two of the key roles. The constraints specified in bold are the ones enacting separate interaction models. For example, the steps 2-3 mentioned above are performed in the constraint *find_path(From,To,Path)* of Figure 7. Such constraint enacts the "Find-Route" IM whose details can be found in appendix A-2.2.2. 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 are performed in the constraint *try_move_action* of the *goal_mover* role[11]; it enacts the "Perform-Action" IM (see appendix A-2.1.7 for more details).

In what follows, we describe in details the activity of checking the path state, since it represents the core part of our simulation. The constraint

[9] In what follows, we will give to "interaction model" the short name "IM".

[10] Here the emergency subordinate ES is what we denoted as moving peer.

[11] This role is fully explained in appendix A-2.2.1.

```
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 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: LCC fragment for the "free-path-finder" role

request_path_state(Path,PathState) of Figure 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 centralised and decentralised 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 interaction. The problem of accessing persistent information acquired during execution of separate interactions is addressed by the OK kernel through a "peer access mechanism" which allows an OpenKnowledge Component[12] (OKC) to access the local knowledge of the peer by invoking methods declared in a specific "PeerAccess" Java class[13].

Figure 8 shows the Java code of the OKC's method associated to the *request_path_state* constraint that implements the activity (a) mentioned before. 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 enaction of a separate interaction model 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 the specific, before 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: Java code for OKC method "*request_path_state*": interaction model enaction

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

[12]More details on OpenKnowledge components can be found in [15].

[13]More details on how to access the peer state can be found in: http://www.few.vu.nl/OK/wiki/doku.php?id=manuals:peer_access.

Centralised Control Behaviour

The centralised 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 a *emergency-subordinate* role in the "evacuation" interaction (see A-2.2.1) and to the *path_info_requester* role of 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=[*Phead* | *Ptail*])
     (b) Check that P is free by interacting with Civil Protection
        i. If the path is free then
           A. move from *Phead* to next location *Ln*
           B. Back to step (b) with P=*Ptail*
        ii. Otherwise back to step (a) to get an alternative path from L to G

- Each ***Reporter*** is subscribed to a *reporter* role in the "Querier-Reporter" IM. It responds to requests for water level information from a querier (e.g., Civil Protection).

Figure 9 schematizes the main interactions between the peers. Full details on "Check_Route_State" and "Querier-Reporter" interaction models are given in appendixes A-2.2.3 and A-2.2.4 respectively.

Decentralised Control Behaviour

The decentralised scenario is characterized by the direct interaction between a moving peer and a suitably selected group of reporters. The behaviour of such peer is as below:

- Each ***Moving Peer*** is subscribed to an *emergency-subordinate* role in the "evacuation" interaction (see A-2.2.1) 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:

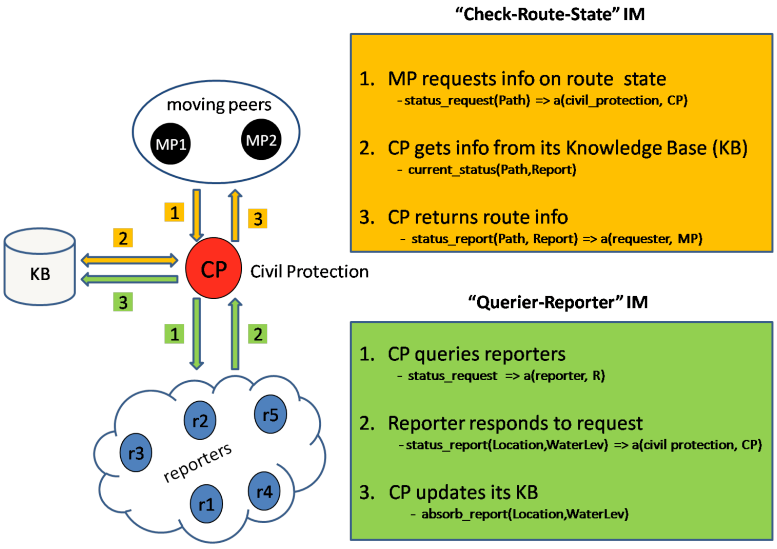

Figure 9: Information Gathering: centralised interactions

1. If $L = G$ then stop
2. Otherwise:
   (a) Get one path P from L to G (P=[$Phead$ | $Ptail$])
   (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 $Phead$ to next location Ln
         – Back to step (d) with P=$Ptail$
      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).

In the above, the key point 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 $P = [L,A,B,G]$. Here, 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 10 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 [16]. 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).

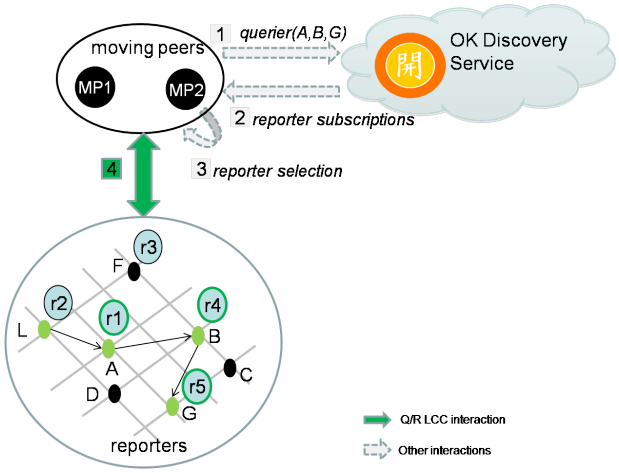

Figure 10: Decentralised Information Gathering: selection of reporters

4.2 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 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 - at present - 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.

4.2.1 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 specific interaction model and (b) it serves requests of performing (move) actions with the "Perform-Action" interaction model (see Figure 4). 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 commences, 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 designed[14]. An LCC code snippet is given in Figure 11. It only shows the key role of the controller. The constraints specified in bold are solved by executing the "Flood", "Sensory-Info" and "Visualiser" interaction models[15] 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 11: LCC fragment for the "info-handler" role taken by the controller

[14]See appendix A-2.1.1 for full details on this interaction model.

[15]The "Flood", "Sensory-Info" and "Visualiser" IMs are fully explained in appendixes A-2.1.4, A-2.1.5 and A-2.1.6 respectively

4.2.2 Flood Sub-Simulator

The flood sub-simulator goal is to simulate a flood in the town of Trento (Italy). The equation defined in its core OKC is based on flooding levels and flooding timings resulted from a flood simulation for the town of Trento, developed by the International Institute for Geo-Information Science and Earth Observation and by the University of Milano-Bicocca [17].

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 year 1966 to 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 two maps for both the water height and the flow velocity. Once such maps are created, they are then transformed into five *indicator maps*, which are shown in Figures 12 through 16. These indicators are:

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

To the purpose of our test-bed, 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 12 and 16. To maintain our simulation realistic but simple, we have assumed that each area reached its maximum flooding level in one hour.

Figures 18 and 17 show a zoom on the north region of Trento. 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 test-bed in order to create two different tables in a geographical database.

Each table has a field, called `node`, representing x,y cordinates of digitalized points. Moreover, the first table has a field, called `MaxWL` (Maximum

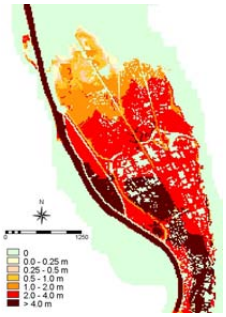

Figure 12: Maximum water level [17]

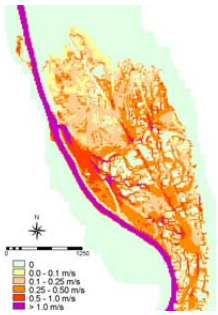

Figure 13: Maximum flow velocity [17]

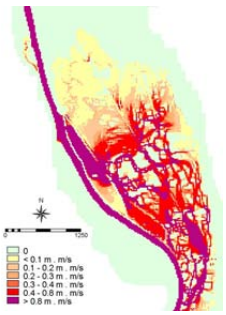

Figure 14: Maximum impulse [17]

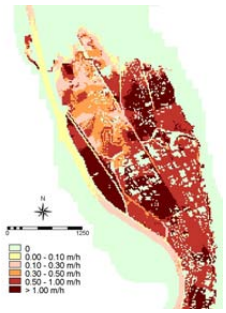

Figure 15: Maximum speed of rising of the water level [17]

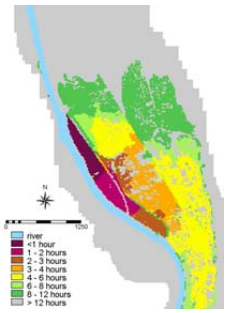

Figure 16: Arrival time of the first flood waters [17]

Water Level), that is the maximum water height for a node. The second table, 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 16) and making the assumption that the time required to culminate the flood is always one hour. Finally, at OK-simulation[16] 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 flooding law used during the OK-simulation to calculate the flood changes for a given node at a time-step t is:

$$\begin{cases} f(t) = 0 & \textit{if } t < (MT-1)*T \\ f(t+1) = f(t) + \frac{(MaxWL)}{T} & \textit{if } (MT-1)*T <= t < MT*T \\ f(t) = f(MT*T) & \textit{if } t >= MT*T \end{cases} \quad (1)$$

[16]We denote our test-bed simulation as the "OK-simulation", in order to distinguish it from the one in [17].

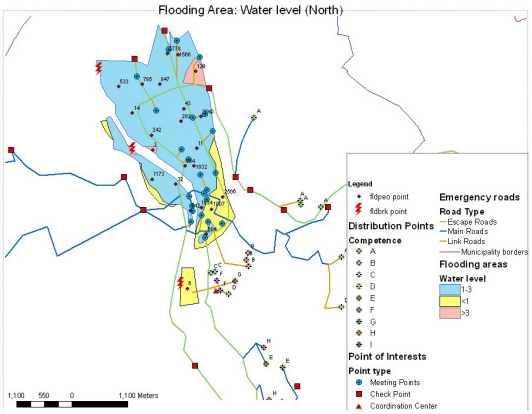

Figure 17: Maximum water level in the north of Trento town

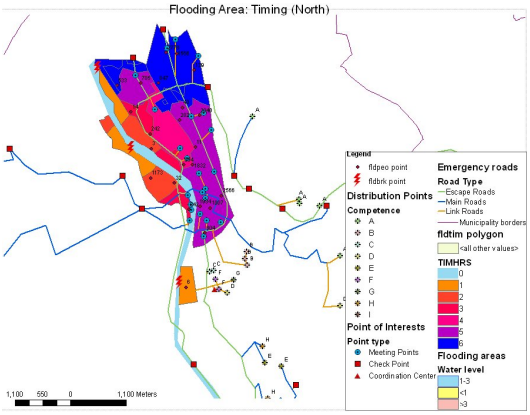

Figure 18: Time when maximum water height is reached in the north of Trento town

where T is the number of time-steps per hour.

In Figure 19, we can see 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. Then, 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.

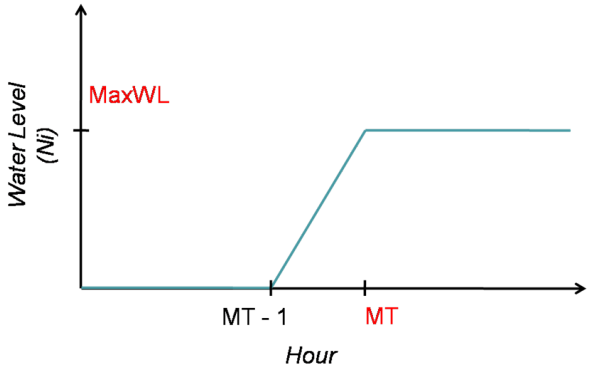

Figure 19: Flooding Law

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, 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 (see A-2.1.2) is enacted

just once at the beginning of the simulation by the *connectWithSubSimulators* constraint in the "Simulation Cycles" IM (see section A-2.1.1). This interaction model has two main goals:

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

The second interaction model (see A-2.1.4) is used by the controller at each time-step, in order to get from the flood sub-simulator the changes of the flood level of 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 (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.

4.2.3 Visualiser

This component enables the GUI used to visualise the simulation. In particular, the GUI shows the information provided by the controller through the "Visualiser" interaction model. At every time-step, the visualiser receives the changes and updates its history according to the new information. The update results in a change on the GUI. Figure 20 shows the appearance of the GUI at the first time-step of the simulation.

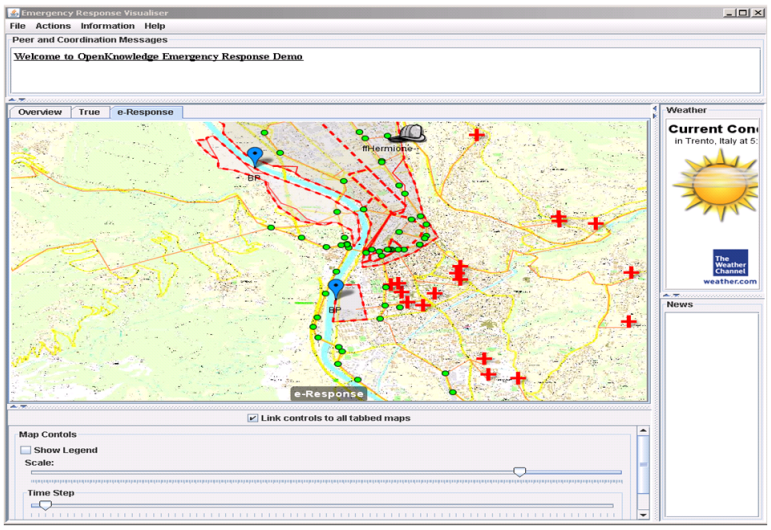

Figure 20: Emergency GUI

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; the hat represents the emergency subordinate.

For more detailed information on the interaction models used to implement the simulator, please refer to the appendix A-2.1.

4.3 Reuse of OpenKnowledge components

To build the e-Response simulation system described in the previous section, we strongly benefit from the possibility of reusing OK components. In particular, the components reused to implement both centralised and decentralised scenarios are LCC specifications and OKC plug-in. Figure 21 shows a complete list of all interaction models implemented for both the pre-alarm and the evavuation phase of the considered use case. It shows also the type of peers involved and the separated interactions called 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 from the table, all the interaction models are used in both centralised (C) and decentralised (D) scenarios, but one: the "Check Route State" IM, which is only used to interact with the central peer CP.

LCC Interaction Models

Phase	Peer Types	N°	Name	Peers	Called IM	IG Strategy
PRE ALARM		1	Sensor Monitoring	WFS, EMS		
		2	Weather Forecast	EMS, S	3	
EVACUATION	Network Peers	3	Evacuation	EC, MP	4,5,6	C/D
		4	Find Route	MP, RS		C/D
		5	Check Route State	MP, CP		C
		6	Perform Action	MP,C		C/D
		7	Querier/Reporter	CP, R		C/D
	Simulator Peers	8	Simulation Cycles	C	9,10,11 ,12,13	C/D
		9	Flood	C,F		C/D
		10	Visualiser	C,V		C/D
		11	Peer Connection	Peer, C		C/D
		12	Sensory Info	C,Peer		C/D
		13	Flood SubSimulator Connection	C, F		C/D

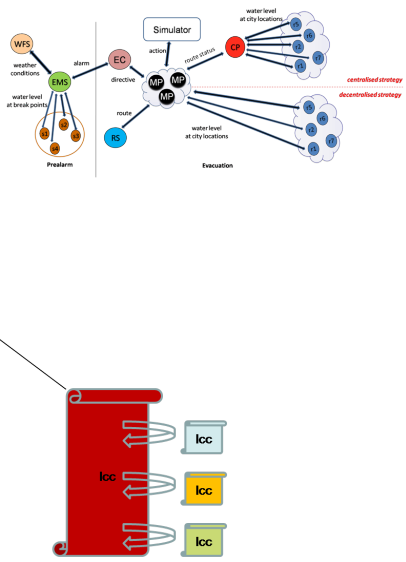

Figure 21: e-Response Interaction Models

The table therefore shows that the interaction models are modular and reusable in different contexts.

Reuse of the "Querier-Reporter"interaction model

From the point of view of the information gathering strategy, the key interaction model is the "Querier-Reporter". Thus, it is interesting to describe the mechanism through which this very same specification is used to enable both centralised and decentralised scenarios (see Figure 22). In the centralised 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 other hand, in the decentralised scenario the moving peer subscribes to the same interaction with the subscription description *querier(Path)* as already described in section 4.1. Finally, the peer *MP* enacts this interaction only when needed, i.e., when it has to move.

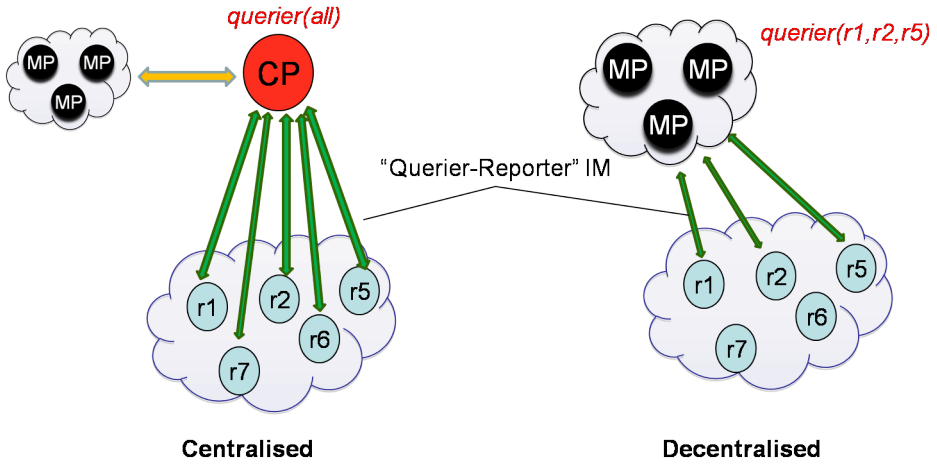

Figure 22: The "Querier-Reporter" IM Reuse

Reuse of OKC methods

Beside reusing the interaction models, the e-Response system implementation was aided by the reuse of OKC components, even though in 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 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. Exploiting the Java's inheritance mechanism is, in fact, possible to define OKC components which will be used by many peer types only once. 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, where needed, it is extended. 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. The former peer will therefore use an OKC component which extends the Java class *ConnectOKC* and contains a method, namely "update_info", that overrides the one defined in the base class and includes the enaction of the "Querier-Reporter" interaction model.

5 The e-Response Summative Experiment

In this section we describe the evaluation of the OK framework in the e-Response domain. We designed a series of experiments with a three-fold aim:

1. Show the OpenKnowledge system in action, illustrating that all parts of the system are capable of working cohesively in the desired manner;

2. Demonstrate that the technology provided by OpenKnowledge supports different models of information sharing (centralised vs decentralised scenario);

3. Establish whether the OpenKnowledge paradigm can make positive differences in performance between such disaster scenarios. In particular, what is expected (and desirable) is to have the OK p2p framework comparable in performance to traditional centralised systems and, when specific fault conditions arise, improving such conventional systems.

While the achievement of the first two objectives depends on an appropriate and efficient design of the OK components (IMs, OKC's), the third goal is less straightforward and hides a certain complexity. Since it is the core of the whole e-Response summative experiment, it deserves a more punctual explanation. We started from the following general evaluation hypotheses:

There exists some combination of interaction model sharing, ontology matching and trust assessment capable in a highly distributed and peer to peer architecture of rivalling the emergency response performance we would expect in a traditional centralised planning model; furthermore, the performance of the peer-to-peer system is more robust in the presence of failure of components.

This general hypothesis is insufficiently precise to be tested directly so we consider a more specific hypothesis:

In simulations using the OpenKnowledge kernel for coordination, Trentino GIS/flood data to represent world state and locational information about people and resources, there exists some combination of OpenKnowledge interaction model sharing, ontology matching and trust assessment capable with a decentralised peer-to-peer system of coordination (relying on the ontology

matching and trust mechanisms) and of moving similar numbers of people to safe sites during a simulated flood event as we observe in simulations with a centralised system of coordination. Furthermore the performance of the peer-to-peer system is more robust to the failure of sensors and breakdown of communication channels.

The above hypothesis is quite plausible but nevertheless compound. In order to investigate it, we built up a framework consisting of four steps: (i) analysis of the variables involved; (ii) determination of meaningful assumptions; (iii) definition of experiments in all their details; (iv) specification of expected results; (v) design of a supporting ICT infrastructure; (vi) experiment execution. These steps will be described in the next sections. However, it is worth to anticipate here that two main types of experiments are designed: experiments simulating emergency scenarios in presence of *ideal* and *fault conditions*.

Once the framework is established and its phases completed, the final task is to run each experiment. In order to prove the hypothesis, 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 current version of the OK kernel is not yet definitively stable with a relative high number of peers and concurrent interaction models. For this reason, for this deliverable, we focused on collecting a significant number of runs of the first type of experiments (centralised and decentralised e-Response scenarios without fault conditions). We will focus on statistically interpretable results for the experiments with faults cases in future work.

5.1 Performance Measurement and Involved variables

In the context of our experiments, what we measure as performance is:

(a) the *percentage of moving peers* arriving at destination;
(b) the *number of timesteps* needed to arrive at destination.

The above indicators will be used to compute the results of the experiments and to make a comparison among them.

As already anticipated, the first step needed to develop the evaluation is to carry out an analysis of the variables involved. Notice that for each experiment designed, a certain number of runs need to be made. A list of the variables considered in the experiments follows:

A. *Number of moving peers*: the number of peers moving to a specific destination. Since the main aim of the summative experiment is to compare two different strategies (centralised vs decentralised) rather than making a realistic simulation, it is reasonable to fix this variable to 1 in all experiments. By running an experiment a certain number of times, we can then compute the performance (a) of the simulated scenario;

B. *Paths*: these are the routes in the topology considered in the experiments. In order to have significant results, it is important to consider, for each experiment type, a meaningful set of routes, that is, routes covering both flooding and non-flooding areas.

C. *Flooding law*: models how the flood evolves over time. The flooding law markedly affects the outcome of an experiment run. For example, the moving peer may either arrive at destination or be blocked dependently on how rapidly the flood propagates along the route taken. In our experiments, the flooding law is fixed and follows the equation 1 of section 4.2.2.

D. *Number of nodes*: locations included in the topology and whose status can be reported by some peer. Incrementing this number is useful to test the capacity of the OK kernel to support many peers. In our testbed, this variable is the number of nodes composing only those routes involved in a given experiment.

E. *Number of (reporter) peers per node*: the number of reporters located in one node. As before, this variable is useful to test the robustness of the OK kernel and, moreover, the effectiveness of some of its modules (e.g., the trust module [18]). Since dedicate experiments already exist which test such modules (see [19] for more details) and given that our summative experiment aims to discover, if some, eventual benefits of a p2p coordination strategy over a more standard centralised one, we fixed this variable to 1.

F. *Degradation of the CPU communication channel*: measured as the likelihood of a fault in the communication channel of the Civilian Protection Unit peer. For example, having a degradation of the 80% means to have this peer serving incoming requests only the 20% of the times. This variable plays a role in the experiments which foresee the presence of inaccurate signaling. In particular, by setting this variable, a specific type of fault (*channel fault*) and its severity can be simulated.

G. *Degradation of reporter communication channels*: defines, for all reporter channels, the probability of their disruption. For example, having a degradation of the 30% means to have each reporter peer serving incoming requests with the likelihood of the 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.

H. *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 assumed all peers were trustworthy.

5.2 Experiment design

A suite of experiments is defined in order to investigate whether the OK framework is capable of supporting emergency evacuation activities which adopt two different models of information sharing (centralised vs distributed strategy). Furthermore, we want to compare the performances of these strategies according to our previously defined indicators. We designed two main classes of experiments:

- ***Experiments with No Fault Conditions***: set of experiments simulating both centralised and decentralised scenarios which evolve under ideal conditions: the absence of faults (e.g., failures in communication, inaccurate signaling) is assumed.

- ***Experiments with Fault Conditions***: set of experiments simulating both centralised and decentralised scenarios where the presence of faults (e.g., failures in communication, inaccurate signaling) is assumed.

Before moving to describe the experiments in more details, it is important to mention here the *assumptions* made to interpret the results in a reasonable way. Moreover, such assumptions are driven by the current number of peers involved and the actual mechanism of the simulation. They are:

I) The Civilian Protection Unit (CPU) 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 and/or communication overloads never occur.

II) A querier, asking a certain number of reporters for information, will receive all the answers within a timestep. This is due to how the timestep interval is set: the value is such that the time elapsing between one timestep and the next one is sufficiently high to guarantee the replies from all the reporters.

By making these assumptions, we simulate a real case scenario where pros and cons of both centralised and decentralised architecture are balanced.

In the next sub-sections, we first describe the experiments without considering the fault conditions. Then, we introduce different types of faults and, finally, we illustrate the experiments where these faults are injected.

5.2.1 Experiments with No Fault Conditions

In this first suite of experiments we assume that there are not faults neither on civil protection communication channel nor on reporters communication channels and that all peers are trustworthy. Since moving peers do not interact (at present), the following experiment settings are equivalent:

(A) Running an experiment only one time with many peers that are moving from different locations to different destinations;

(B) Running many times each experiment with only a peer that is moving from a different location to a different destination at each run.

Below we describe how some of the previously defined variables were instantiated, adopting the experiment setting (B). In this case, centralised and decentralised simulations have the same configuration apart the experiment type.

- *Number of moving peers (A)*: one moving peer per run;
- *Paths (B)*: at each run the moving peer has to cover a different distance;
- *Flooding law (C)*: the equation is fixed;
- *Number of nodes (D)*: we don't have a reporter peer on each node of the topology, but we locate, at each run, 70 reporters in different nodes;
- *Number of (reporter) peers per node (E)*: one reporter peer per node.

			Variable Settings				
Exp N°	**Information Gathering**	**Runs**	**A**	**B**	**C**	**D**	**E**
1	centralised	10	1	1 distance x run	fixed	70 x run	1
2	decentralised	10	1	1 distance x run	fixed	70 x run	1

Table 1: Experiments configuration (no fault conditions)

Table 1 summarizes the experiment configuration: each experiment is run 10 times; at each run, the only variables that change are the distances that should be covered by the moving peer and the locations where reporters are present. Such locations are determined according to the set of routes associated with the destination assigned to the peer and its starting position. The flooding law, the number of emergency subordinates and the number of reporters remain unchanged during all runs.

Running the above experiments and under the assumptions (I) and (II), we expect that the results we obtain are similar and therefore we should be able to conclude that the OpenKnowledge framework is capable of supporting centralised and decentralised architectures with comparable performance.

5.2.2 Introducing fault conditions

We constructed a basic fault tree analysis for the e-Response simulation in order to identify the types of failures and their relation with events that can occur in our simulation. We did not associate probabilities to faults in order to estimate likelihoods of these events; rather, we performed this analysis only as a means to guide us in the experiment design. In our simulations, we considered two primitive faults:

- *Communication failures*: the channels are broken;
- *Inaccurate signaling*: the water level is inaccurately reported.

In Figure 23, a possible fault tree[17] is shown. In the graph, primitive faults (represented as rectangles) are applied to two different initial conditions (represented as circles): safe water level or unsafe water level. In the first case, an inaccurate information reporting results from a sensor signaling an unsafe water level, while, in the second case, it is due to a sensor signaling a safe water level. Starting from the right upper part of the graph, i.e., from the "Sensor fails to communicate" fault and the "Unsafe water level" top condition, the bottom event "Person does not reach safe area" is deduced from the intermediate "No hazard signal" condition and the "Person guided to unsafe area" event. The above depicted path represents the case in which, if the water is at unsafe level and the communication channel is broken, the person may not reach a safe location.

To simulate these faults we changed the behavior of *reporter* and *civil protection peers*, that is, we changed the Java methods in the related OKC components. An inaccurate information reporting from a sensor peer can be simulated by introducing some noise in the real water level received from the simulator as described in [19]. It can be noticed that this fault is restricted to sensor peers, since we assume that, in the centralised scenario, the CP is always trustworthy. A broken channel, instead, can be simulated for both sensor and CP peers by not sending a response message when the peer in question is queried. The faults above can occur with different frequencies: their probabilities are modeled using the variables described in section 5.1 (*Distribution of trustworthy (reporter) peers* , *Degradation of the CPU communication channel* and *Degradation of reporter communication channels*).

5.2.3 Experiments with Fault Conditions

In our overall work in the e-Response scenario, we have considered two fault conditions, namely: (1) inaccurate and false signaling form the reporters and (2) degradation of communication channels. The inaccurate signaling fault has been explored in detail in the evaluation of the Trust component in Deliverable 4.9 [19]. The main result there is that the use of the OK

[17]The fault tree has been proposed by Dave Robertson.

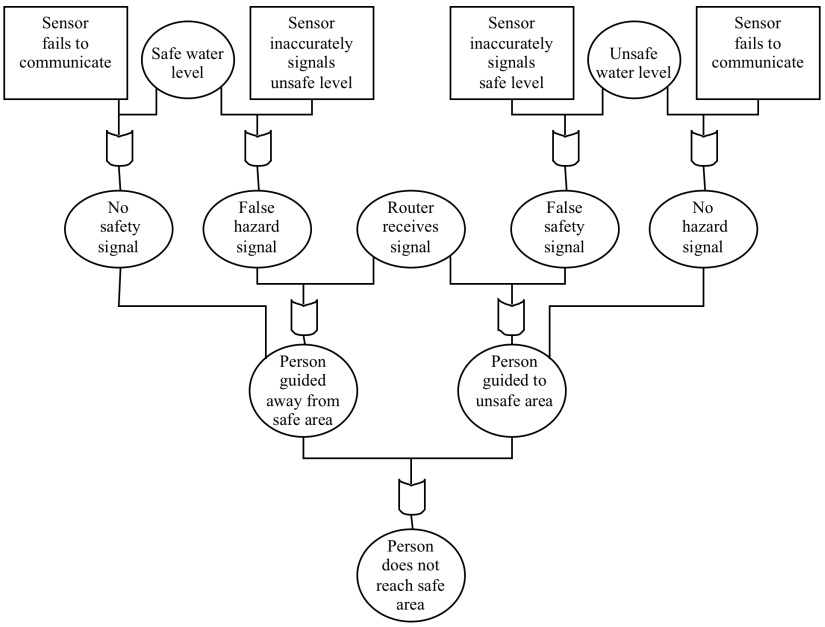

Figure 23: Basic fault tree graph for e-Response simulation

framework, and specifically the OK Trust component, provide a relevant improvement in the performance of the selection of the most reliable peers to interact with.

Here, we have focused our attention on the second type of fault condition, i.e. degradation of communication channels. To this end, we have designed the following four types of experiments:

- ***Experiment 1***: centralised scenario with perfect CPU and fixed degradation of sensor communication channels;
- ***Experiment 2***: centralised scenario with CPU channel degraded at 30% and fixed degradation of sensor communication channels;
- ***Experiment 3***: centralised scenario with CPU channel degraded at 80% and fixed degradation of sensor communication channels;
- ***Experiment 4***: decentralised scenario with a fixed degradation of sensor communication channels.

Figure 24 facilitate to visualise the experiments described above. In particular, it shows where the communication channel faults could be located.

Also for this suite of experiments, we adopt equivalence (B) and we configure variables as described in section 5.2.1. Here, however, we consider two more variables: *Degradation of the CPU communication channel* (F) and *Degradation of reporter communication channels* (G).

Table 2 summarizes the experiment configuration when a broken channel fault is injected into the system. In this table, *Number of moving peers*,

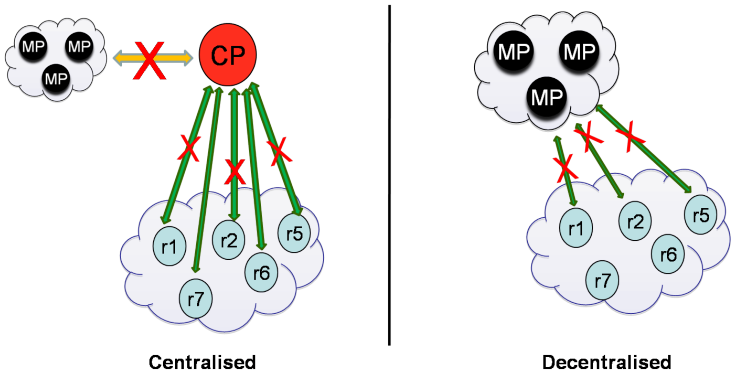

Figure 24: Communication channel faults: centralised and decentralised strategy

			Variable Settings	
ExpN°	**Info Gathering**	**Runs**	**F**	**G**
1	centralised	10	0%	30%
2	centralised	10	30%	30%
3	centralised	10	80%	30%
4	decentralised	10	not applied	30%

Table 2: Experiments configuration (with fault conditions)

Paths, *Flooding law*, *Number of nodes* and *Number of (reporter) peers per node* variables are not shown since they are instantiated as described in section 5.2.1. Briefly we remark that each experiment is run 10 times and that, at each run, only the *Paths* variable changes.
Regarding the new variables (F and G), we designed the experiment by changing only the probability of breakdowns in the CP communication channel since we want to explore how severe is the impact of faults at the centralising element. If we also want to explore the impact of faults in the sensor communication channels, we have to repeat the above defined experiments modifying the variable G.

5.3 Experiment setup and launch

In this section we present details of the architecture used to run experiments for the e-Response testbed. In order to run an experiment, a process for each involved peer needs to be launched. For this purpose, we developed a Java program that reads the selected configuration variables from a database and then launches the processes with different combination of parameters. Some features of the peer processes are dynamically set up at run time, reading configuration parameters from a database, according to the case we want to test. This mechanism exploits the distributed nature of the OK platform. For example, while the Discovery Service (DS) [16] is run in one server, the processes associated with the reporter peers are launched in a different machine. Peers involved in the experiment are:

- ***Discovery Service***: it is an OK infrastructure peer. We launch only one DS peer.

- ***Simulator***: it starts the whole simulation and it plays the controller role in many interaction models. For each experiment run we launch one simulator peer. When we launch this peer we set up the following parameters that are used in the initialization phase of the *Simulation Cycles* interaction model (see A-2.1.1):

  - *Experiment_id*: it identifies the current e-Response experiment. It is used to read configuration settings and to store results in the database;
  - *Run count*: it identifies the particular run number for the above specified e-Response experiment;
  - *Max number of simulation cycles*: it represents the duration of the simulation. It has to be sufficiently high to guarantee that all moving peers have enough time-steps to reach their destinations;
  - *Expected number of peer connections and average peer connection time*: these two parameters are used to calculate the maximum amount of seconds the simulator has to wait for connecting peers before reaching a connection timeout and going on with the execution of the interaction model;
  - *Water level threshold*: this value sets the minimum water level beyond which a road is considered blocked. It strongly affects the outcome of an experiment run.

- ***Flood sub-simulator***: it is the peer that simulates the flood evolution event in Trento. We only have one flood sub-simulator peer. At run time we dynamically set up a parameter in the flooding law implementation that regulates the flood evolution rate for the current simulation: we decide how many time-steps correspond to one hour in the real world.

- ***Emergency Chief***: it starts the evacuation phase by sending to a moving peer the directive to go to a specific destination. This goal, for each moving peer involved in the current run, is set up dynamically reading the experiment configuration in the database. We have only one peer of this class.

- ***Moving peer***: it is the peer that during the simulation goes from a starting node to a specific destination. In all experiments we ran we have only a moving peer but the database schema and the Java program are designed to launch any number of peers in parallel. This program is also used to configure other parameters like the peer name, the peer selection strategy (randomly, trust score based, user based,

etc.), its initial position and the experiment type (i.e. centralised or decentralised).

- ***Route Service***: it provides a route that connects two given locations. We have only one peer of this class.

- ***Civil Protection***: it gives information on the blockage state of a given path. For this peer we dynamically set up the peer selection strategy and the state of the communication channel. In the suite of experiments run, the channel is always ideal (with infinite band and without breakdowns), but the database is designed to have any probability of errors in the transmission. We have only one peer of this class.

- ***Reporters network***: it is a collection of peers that gives information on the water level registered at their location. In our experiments it turns to be a service peers network, i.e., a set of sensors permanently placed at a location. We launch about 70 sensor peers using the methodology described in [19].

5.3.1 Database description

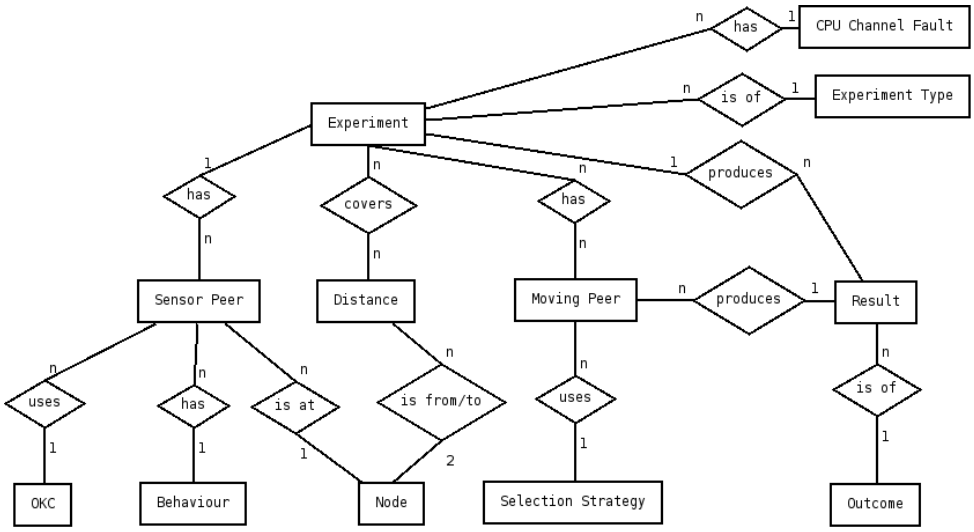

Figure 25: Entity-Relationship diagram for the e-Response experiment

The entity-relationship (ER) diagram of Figure 25 represents the overall logical structure of the database used in the e-Response summative experiment. It shows only the most meaningful entities and relationships.

The most important entity is ***experiment***. It is defined by a description, the total number of runs for each experiment, the total number of sensor peers that will be launched, the maximum duration of the simulation expressed in

time-steps and the number of distances that will be covered during experiments runs. As you can see from the following SQL query, if its value is one, then all peers in all runs cover the same distance, otherwise at each run they cover a different one.

```
SELECT to_node FROM distance WHERE distance_id IN
       (SELECT CASE
               WHEN covered_distances = 1 THEN 1
               ELSE   runNumb
       END
       FROM experiment WHERE sim_exp_id = expID )
```

The ***distance*** entity is represented by the starting node, the destination node, the number of possible paths from these two nodes, the minimal path length, expressed in number of nodes gone through, and the minimal path distance, expressed in meters. This object is related to the ***node*** entity that has x and y geographical coordinates attributes. The coordinate reference system is UTM-WGS84. Table 3 shows the physical implementation of the distance table with some example records.

distance id	from node	to node	number of paths	minimal path length	minimal path distance
1	177	8	12	28	5630
2	177	100	10	18	2192
3	177	87	12	36	7531
4	177	76	12	26	3701
5	76	310	24	20	5174
6	188	245	6	19	4916
7	51	346	6	20	4292

Table 3: Records from the distance physical table

An experiment has also a ***type*** that describes how peers share information. A type could be *centralised* or *decentralised*. As you can see from the above ER diagram, the same type can be taken by many experiments.

An other many-one relationship is with ***CPU channel fault*** entity. This entity describes the probability of channel breakdown at CPU side. For the current experiment we defined three types of channel fault: a $perfect\ channel$, with an error probability of 0%; a $low\ channel\ fault$, with an error probability of 30%, and a $high\ channel\ fault$, with an error probability of 80%. In Table 4 you can see the physical experiment table with some records.

exp id	description	run	type id	channel fault id	distances	peers	time-steps
5	Centralised	10	2	1	10	73	50
6	Decentralised	10	3	1	10	73	50
11	InaccurateInfo_CentDecent	10	4	1	1	292	80
13	Decentralised_CHFault	10	3	1	1	73	80
121	Centralised_CHFault_PerfectCP	10	2	1	1	73	80
122	Centralised_CHFault_LowCPError	10	2	2	1	73	80
123	Centralised_CHFault_HighCPError	10	2	3	1	73	80

Table 4: Records from the experiment physical table

Many peers join an experiment. First of all, there are one or more ***moving peers*** that should go from an initial location to a destination. Each

moving peer is represented by a name and is related to the ***selection strategy*** entity. This entity represents the way in which a moving peer selects a reporter peer in the decentralised scenario. In our experiments, we used always the same selection strategy but experiments were carried out where different strategies are tested [19]. In Table 5 you can see how the many-to-many relationship between the experiment entity and the moving peer entity is phisically implemented in the database.

id	exp id	peer id
5	5	1
6	6	23
7	11	2
9	11	4
10	11	5
11	11	6
12	11	7

Table 5: Records from the experiment_moving_peer physical table

Sensor peers also join an experiment. They are located at a node and they send information about the flooding level.
Each sensor peer has a ***behavior***, i.e., the error that will be added to the real water level values given by the *flood sub-simulator*. The behavior could be *Correct* behavior, with an error of 0, or *Incorrect* behavior, with an error of 0.9. In the e-Response experiments we use only sensor peers with correct behavior, see [19] for the description of the other cases.
Moreover, a sensor peer has an OKC. The ***OKC*** entity defines how a sensor peer will satisfy the constraint defined in an interaction model. For the purpose of our experiment we always use the same OKC class, that has methods that match perfectly the corresponding constraints in the interaction model subscribed. Differently, in project Deliverable 4.9 [19] we used different OKC classes since we wanted to test the OpenKnowledge Matcher module [20]. In Table 6 the physical implementation of the sensor peer table with some example records is shown.

peer id	peer name	nodeid	behavior id	okc id	exp id
1024	Correct_node2_peer1024_okc1e1.0exp8.com	2	1	1	8
1025	Correct_node3_peer1025_okc1e1.0exp8.com	3	1	1	8
1026	Correct_node4_peer1026_okc1e1.0exp8.com	4	1	1	8
1027	Correct_node6_peer1027_okc1e1.0exp8.com	6	1	1	8
1028	Correct_node9_peer1028_okc1e1.0exp8.com	9	1	1	8
1029	Correct_node10_peer1029_okc1e1.0exp8.com	10	1	1	8
1030	Correct_node12_peer1030_okc1e1.0exp8.com	12	1	1	8
1031	Correct_node14_peer1031_okc1e1.0exp8.com	14	1	1	8

Table 6: Records from the sensor physical table

Finally, the ***result*** entity models the outcome of an experiment run for a particular moving peer. In the ER diagram you can note, in fact, that it is related to the experiment entity, the moving peer entity and the outcome

entity. In our experiment runs the outcome is simply either *arrived* or *not arrived*. This entity is also defined by two attributes indicating respectively when an experiment run starts and ends, by putting the effective number of nodes gone through to reach the destination and how many time-steps were needed. These two last attributes are very interesting because they can be used to analyse performance run. For example we can compare the effective number of nodes gone through to the minimal path length of the corresponding distance. We can also compare the total number of time-steps needed to reach the destination to the respective value of others runs that cover the same extent.
Since at run time, for the same experiment configuration, we can set at which water level a road is blocked, we also have a water level threshold in result's attributes. The last attribute is the amount of time-steps missed during a run. This attribute was used to analyse the OpenKnowledge kernel robustness. In Table 7 the physical implementation of the sensor peer table with some example records is shown. Unlike previously described tables, this table is populated at run time: some values are inserted at the beginning of an experiment run, some others at its end.

resultId	expId	run	peerId	outcomeId	starting time	ending time	pathLength	Tstep	WL	missedTS
159	6	1	23	1	2008-11-27 11:51:33	2008-11-27 12:03:11	28	29	0.8	0
165	6	2	23	3	2008-11-27 15:55:19	2008-11-27 16:02:08	17	18	0.8	0
170	6	3	23	1	2008-11-27 18:23:12	2008-11-27 18:38:42	36	37	0.8	0
171	6	4	23	3	2008-11-27 22:26:47	2008-11-27 22:34:57	17	18	0.8	0
174	6	5	23	3	2008-11-28 10:18:34	2008-11-28 10:31:06	30	31	0.8	0
175	6	6	23	1	2008-11-28 10:34:46	2008-11-28 10:43:12	19	20	0.8	0
177	6	7	23	1	2008-11-28 11:17:59	2008-11-28 11:26:21	20	21	0.8	0
183	6	8	23	1	2008-11-28 14:08:00	2008-11-28 14:13:12	11	12	0.8	0
185	6	9	23	1	2008-11-28 14:50:36	2008-11-28 15:04:21	30	31	0.8	0
196	6	10	23	1	2008-11-29 11:36:23	2008-11-29 11:47:33	26	27	0.8	0

Table 7: Records from the result physical table

5.4 Experimental results

The experiments we have run for the e-Response test-bed are the ones considered in section 5.2.1, i.e., the experiment designed without any fault condition.

Each experiment consists in the simulation of the evacuation scenario described in the previous section. Independently on the kind of strategy adopted, the final goal of an emergency subordinate is to safely reach the assigned destination. There are three situations which may happen: (1) the agent reaches the destination by following the first route found; (2) the agent finds blocked routes but finally reaches the destination after a number of alternative paths and (3) the agent does not reach the destination at all. We refer to each situation as the *outcome* of the experiment.

We run each experiment 10 times. The simulations were visualized on the GUI in order to analyze the movements of the emergency peers and verify the correct mechanism in the coordination among the agents.

Figures 26 and 27 show a simulation run for the centralised and the decentralised scenario respectively. Figure 26 shows the agent out from the flooded area. Here, all the dots are grey, meaning that all reporters are being queried by the Civil Protection in order to obtain the water level of their location. Some of them register high levels of water.

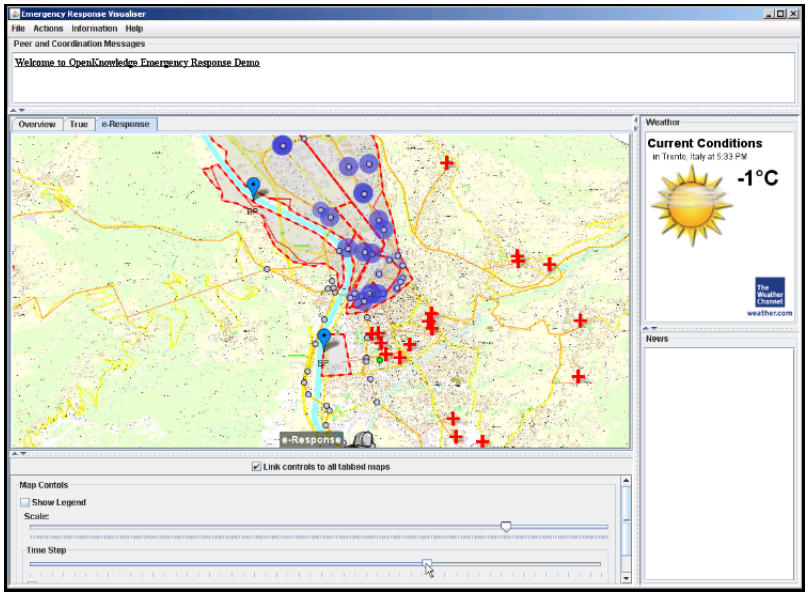

Figure 26: GUI: Centralised information gathering

Figure 27 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). Here, the OK paradigm is exploited in its decentralised nature, since the information gathering is based on the use of distributed information reporter agents and not on an unique provider, as in the first case.

Figure 28 shows the outcome distribution obtained by running 10 times the first experiment. As can be seen, 70 percent of the times, the experiment has outcome (1) (the peer reaches the destination without problems) while 30 percent 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 decentralised 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

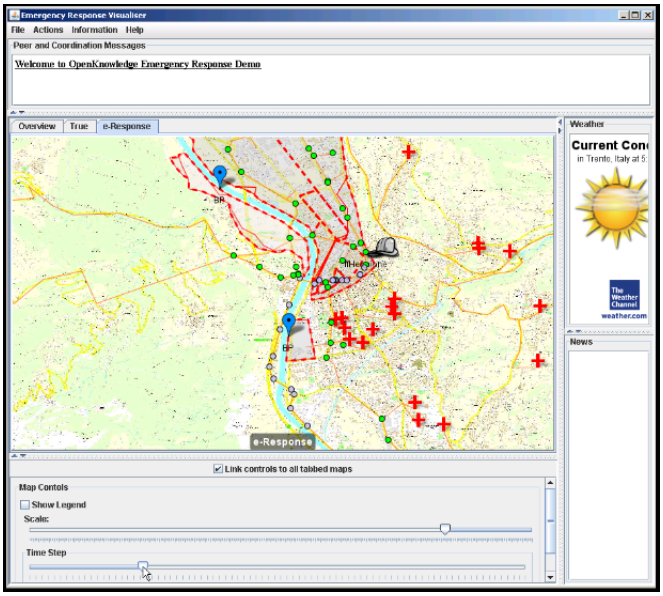

Figure 27: GUI: Decentralised information gathering

Civil Protection or reporters scattered around the city does not make the difference.

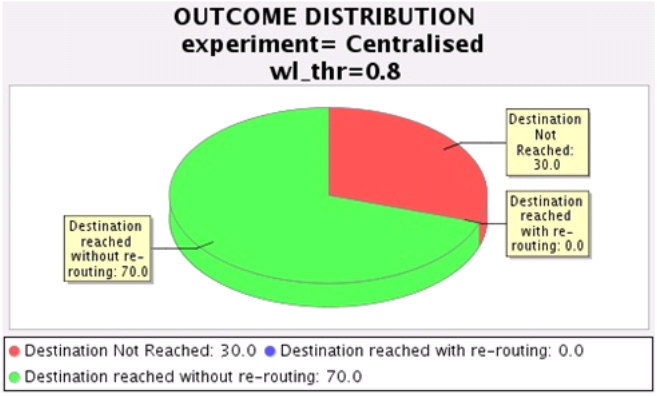

Figure 28: Outcome Distribution (centralised/decentralised scenario)

Figure 29 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 how the simulation is designed - an agent moves from a location to the next one exactly in a time-step - and by the missing outcome (2). Finally, Figure 29 reveals very similar trends for both centralised and decentralised scenarios. Again, this is mainly due to the assumptions made and the variable settings.

In view of the results described above, we can conclude that our first

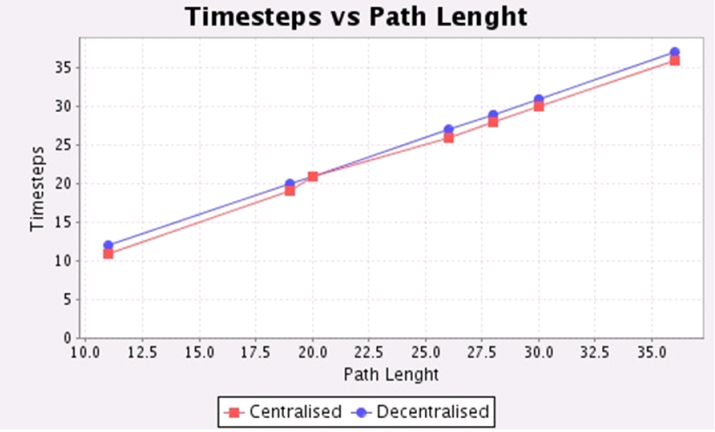

Figure 29: Time-steps vs. Path Length

expectation is met: the use of the OK framework supports both architectures (centralised and decentralised) and provides comparable performances under the selected - ideal - assumptions.

6 Conclusions and Future work

Our work focused on testing whether the OK framework is capable to support the coordination of emergency activities and how, in absence of fault conditions, the OK p2p framework is comparable in performance to traditional centralised gathering approaches. An agent-based e-Response simulation system fully integrated with the OpenKnowledge infrastructure has been developed. This system currently runs on Java and exploits the distributed nature of the OK platform. It is used to model specific emergency scenarios and agents in terms of both LCC specifications and OKC components. A suite of experiments has been designed and run to evaluate the performance of the OK e-Response system in different scenarios and under specific assumptions. The preliminary results thus obtained show how the OK infrastructure is equally effective in both centralised (hierarchical) and decentralised (p2p) information gathering.

We are currently working on further experiments. In particular, we want to repeat the reported experiments both increasing the number of runs and tuning parameters like the "flood speed" and the routes to follow, in a way that more varied outcomes can be obtained. In this way, we could reconfirm our hypothesis in a more robust setting. Also, we want to run experiments where the communication channel faults described in section 5.2.2 are injected. What we want to investigate by running these experiments with communication faults, is if - and eventually under which conditions - a complete p2p architecture improves the overall performance and robustness over traditional centralised architectures.

Finally, from the point of view of the simulated scenarios and the involved

agents, it would be interesting to consider the reporter agents as mobile emergency agents rather than fixed sensors. In this way, we could explore how the OK platform supports the coordination of team-members in an emergency site.

7 Acknowledgments

We are thankful to Dave Robertson for his advices on the formulation of the hypothesis to test. We are much grateful for the constant support of Paolo Besana, who gave us numerous feedbacks on the OK kernel. Also, we are grateful to David Dupplaw for the development of the emergency GUI, to Juan Pane who developed the database we further extended and to Fiona McNeill who developed the initial prolog controller that we adapted to the OK platform.

References

[1] Lorincz, K., Malan, D., Fulford-Jones, T., Nawoj, A., Clavel, A., Shnayder, V., Mainland, G., Welsh, M., Moulton: Sensor networks for emergency response: challenges and opportunities. Pervasive Computing **3** (2004) 16–23

[2] Mecella, M., Catarci, T., Angelaccio, M., Buttazzi, B., Krek, A., Dustdar, S., Vetere, G.: 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)

[3] D'Aprano, F., de Leoni, M., Mecella, M.: Emulating mobile ad-hoc networks of hand-held devices. the octopus virtual environment. In: Proceedings of the first ACM Workshop on System Evaluation for Mobile Platform: Metrics, Methods, Tools and Platforms (MobiEval) at Mobisys. (2007)

[4] Robertson, D.: A lightweight coordination calculus for agent systems. Lecture Notes in Computer Science - DALT **3476** (2005) 183–197

[5] Bellamine-Ben, S.N., Dugdale, J., Pavard, B., Ahmed, M.B.: 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)

[6] Murakami, Y., Minami, K., Kawasoe, T., Ishida, T.: Multi-agent simulation for crisis management. In: Proceedings of the IEEE International Workshop on Knowledge Media Networking. (2002)

[7] Bellamine-Ben, S.N., Mena, T.B., Dugdale, J., Pavard, B., Ahmed, M.B.: Assessing large scale emergency rescue plans: an agent based approach. special issue on emergency management systems. International Journal of Intelligent Control and Systems **11** (2006) 260–271

[8] Kanno, T., Morimoto, Y., Furuta, K.: A distributed multi-agent simulation system for the assessment of disaster management systems. International Journal of Risk Assessment and Management **6** (2006) 528–544

[9] Massaguer, D., Balasubramanian, V., Mehrotra, S., Venkatasubramanian, N.: Multi-agent simulation of disaster response. In: Proceedings of the First International Workshop on Agent Technology for Disaster Management. (2006)

[10] Helin, H., Klusch, M., Lopes, A., Fernandez, A., Schumacher, M., Schuldt, H., Bergenti, F., Kinnunen, A.: 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)

[11] Han, L., Potter, S., Beckett, G., Pringle, G., Sung-Han, K., Upadhyay, R., Wickler, G., Berry, D., Welch, S., Usmani, A., Torero, J., Tate, A.: Firegrid: An e-infrastructure for next-generation emergency response support. In: Submitted to Royal Society Phil. Soc. A. (2009)

[12] Vaccari, L., Marchese, M., Giunchiglia, F., McNeill, F., Potter, S., Tate, A.: *OpenKnowledge Deliverable 6.5: Emergency response in an open information systems environment.* `http://www.cisa.inf.ed.ac.uk/OK/Deliverables/D6.5.pdf` (2006)

[13] Vaccari, L., Marchese, M., Shvaiko, P.: *OpenKnowledge Deliverable 6.6: Emergency Response GIS Service Cluster.* `http://www.cisa.inf.ed.ac.uk/OK/Deliverables/D6.6.pdf` (2006)

[14] Marchese, M., Vaccari, L., Trecarichi, G., Osman, N., McNeill, F.: Interaction models to support peer coordination in crisis management. In: 5th International Conference on Information Systems for Crisis Response and Management. (2008)

[15] de Pinninck, A.P., Dupplaw, D., Kotoulas, S., Schorlemmer, M., Siebes, R., Sierra, C.: *OpenKnowledge Deliverable 1.2: Peer to peer coordination protocol.* `http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D1.2.pdf` (2006)

[16] Kotoulas, S., Siebes, R.: *OpenKnowledge Deliverable 2.2: Adaptive routing in structured peer-to-peer overlays.* `http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D2.2.pdf` (2007)

[17] Alkema, D., Cavallin, A., Amicis, M.D., Zanchi, A.: Valutazione degli effetti di un alluvione: il caso di trento. Studi Trentini di Scienze Naturali : Acta Geologica **78** (2003) 55–62

[18] Pane, J., Sierra, C., de Pinninck, A.P., Shvaiko, P.: *OpenKnowledge Deliverable 4.8: Plug-in component supporting trust.* `http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D4.8.pdf` (2007)

[19] Pane, J., Sierra, C., Trecarichi, G., Marchese, M., Besana, P., McNeill, F.: *OpenKnowledge Deliverable 4.9: Summative report on GEA, trust and reputation: integration and evaluation results.* `http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D4.9.pdf` (2008)

[20] Yatskevich, M., Giunchiglia, F., McNeill, F., Shvaiko, P.: *OpenKnowledge Deliverable 3.4: Ontology matching component.* `http://www.cisa.informatics.ed.ac.uk/OK/Deliverables/D3.4.pdf` (2007)

A-1 Prealarm Interaction Models

This appendix includes technical details for the two interaction models used to simulate the prealarm phase.

A-1.1 Sensor_monitoring.lcc

The "Sensor_monitoring" interaction model is used by the Emergency Monitoring System (EMS) peer to constantly monitor the flood level in critical points along the river. To do that, the EMS interacts with water level sensors placed in these points. Also, the EMS checks weather information[18] (e.g., precipitation values) in order to enrich the data needed to predict the evolution of a potential flooding. When the EMS registers a critical situation (e.g., water level above a certain threshold and heavy rain), it notifies the emergency coordinator who is in charge of taking the proper actions. As can be seen below, the main roles of this interaction model are *emergency_monitoring_system*, *sensor* and *emergency_coordinator* which are played respectively by the EMS, a water level sensor and the emergency coordinator (EC). The EMS, which begins the interaction, starts by initiating the parameters that will be used during the interaction. These are the following and are needed to set:

- *Timestamp*: the time of the next polling cycle;
- *SPL*: the list of peers playing the role of "sensor";
- *UnitMeasure*: in what unit of measure a water level sensor must return the registered value;
- *SleepTime*: the time elapsing between two consecutive polling cycles;
- *ExpireTime*: the maximum time needed to wait for incoming messages. When this time expires the receiver continue the interaction;
- *SleepTime_WF*: time waited by the EMS in order to ensure the most updated weather information;
- *MaxCycle*: the maximum number of polling cycles. This parameter is set in order to stop the interaction;
- *NoOfCycle*: the current polling cycle; by checking this paramter it can be deduced whether the interaction will terminate or not.

Once the above parameters are instantiated, the EMS jumps to the role *emergency_monitoring_system1*.

[18] The weather information are periodically retrieved in parallel to this interaction. They are acquired through the "Weather_forecast" interaction explained in the next section.

```
r(emergency_monitoring_system,initial)
r(emergency_monitoring_system1,auxiliary)
r(wl_request_sender,auxiliary,1,4)
r(risk_evaluator,auxiliary)
r(riskmsg_sender,auxiliary)
r(stop_sender,auxiliary)
r(sensor,necessary,1,4)
r(emergency_coordinator,necessary)

a(emergency_monitoring_system,EMS)::

 null <- getInitialTime(Timestamp) and getUnit(UnitMeasure) and
         getSleepTime(SleepTime) and getPeers(''sensor'',SPL) and
         setMaxCycles(MaxCycle) and setCycleCounter(NoOfCycle) and
         setExpireTime(ExpireTime) and getSleepTimeWF(SleepTime_WF) then

  a(emergency_monitoring_system1(SleepTime_WF,SPL,ExpireTime,Timestamp,
                                 UnitMeasure,MaxCycle,NoOfCycle,SleepTime),EMS)
```

The LCC code relative to the *emergency_monitoring_system1* role represents the core part of the interaction model. Here, the EMS realizes the polling cycle: it first gets, through the constraint *"getForecast"*, the weather information updated to the current time; if they are not yet available, it waits till they are acquired (*sleepWF* constraint); it then asks all sensor peers for water level values (*wl_request_sender* role), hence receives their replies and evaluate potential risks (*risk_evaluator* role); finally, it eventually sends an alarm to the emergency coordinator (*rskmsg_sender* role). At this point, some parameters are updated and the cycles starts again unless the maximum number of cycles has been reached. Among the updated parameters, there is the *RiskList* variable which contains, for each sensor, the level of risk associated with its location. Also, before leaving its role, the EMS sends a stop message to all sensors (*stop_sender* role) and the emergency coordinator[19].

```
a(emergency_monitoring_system1(SleepTime_WF,SPL,ExpireTime,Timestamp,
                  UnitMeasure,MaxCycle,NoOfCycle,SleepTime),EMS)::

   (
     null <- getForecast(Timestamp,Precipitation) and size(SPL,N) then
       a(wl_request_sender(SPL,Timestamp,UnitMeasure),EMS) then
         a(risk_evaluator(N,ExpireTime,SleepTime,Precipitation,RiskList),EMS)
          then a(riskmsg_sender(RiskList),EMS)
            then
                 (
                   null <- sleep(SleepTime) and
                            updateTime(Timestamp,NewTimestamp) and
                            updateRiskList(RiskList) and
                            inc(NoOfCycle,NewNoOfCycle) and
                            less(NewNoOfCycle,MaxCycle) and
                            updatePrecList(Precipitation) then

                   a(emergency_monitoring_system1(SleepTime_WF,SPL,
```

[19]The sending of stop messages is made just to let the other peers leaving their respective roles. In a real situation one could think to have the sensors and the emergency coordinator always in a "listening mode"

```
                    ExpireTime,NewTimestamp,UnitMeasure,
                    MaxCycle,NewNoOfCycle,SleepTime),EMS)
                )
                or
                (
                  a(stop_sender(SPL),EMS)  then
                    stop => a(emergency_coordinator,EC)
                )
  )

  or

  (
    null  <-  sleepWF(SleepTime_WF) then
     a(emergency_monitoring_system1(SleepTime_WF,SPL,ExpireTime,Timestamp,
        UnitMeasure,MaxCycle,NoOfCycle,SleepTime),EMS)
  )
```

The LCC code for the *wl_request_sender* role is expressed below. Once in this role, the EMS scans the list *SPL* of the sensors and sends, to each of them, a *water_level_request* message. The two parameters *Timestamp* and *UnitMeasure* are specified in the message.

```
a(wl_request_sender(SPL,Timestamp,UnitMeasure),EMS) ::

 null <- SPL=[]

 or

 (
  null <- SPL=[H|T] then
   water_level_request(Timestamp,UnitMeasure) => a(sensor,H) then
    a(sender(T,Timestamp,UnitMeasure),EMS)
 )
```

After having sent all the requests, the EMS takes the *risk_evaluator* role by which: (i) the replies from the sensors are received through the incoming message *water_level*; (ii) the water level data are stored in a local DB (*store* constraint); (iii) the risk associated to each sensor location is evaluated (*evaluate_risk* constraint). Notice that, in the evaluation process, the precipitation rate is also considered together with the water level value. At the end of this role, a list *RiskList* containing all the computed risks is created. Finally, besides the above mentioned activity, the role comprises a part that models the time elapsing. This is done by checking and decrementing the variable *ExpireTime*. Once a unit time has elapsed, the role starts again through recursion. The LCC code relative to the just described role is shown below.

```
a(risk_evaluator(N,ExpireTime,SleepTime,Prec,RiskList),EMS)::

 null <- equalZero(N)

 or

 (
  water_level(Timestamp,SensorName,UnitMeasure,WaterLevel)
   <= a(sensor,S) then
       null <- store(Timestamp,SensorName,WaterLevel) and
               evaluate_risk(UnitMeasure,WaterLevel,SensorName,Prec,Risk) and
               dec(N,NewN) and addList(N,Risk,SensorName,RiskList) then

     a(receiver_risk_evaluator(NewN,ExpireTime,SleepTime,Prec,RiskList),EMS)
 )

 or

 (
  null <- equalZero(ExpireTime)

  or

  (
   null <- sleep(SleepTime) and dec(ExpireTime,NewExpireTime) then
    a(receiver_risk_evaluator(N,NewExpireTime,SleepTime,Prec,RiskList),EMS)
  )
 )

 or

 (
  null <- sleep(SleepTime) then
   a(receiver_risk_evaluator(N,NewExpireTime,SleepTime,Prec,RiskList),EMS)
 )
```

Once all (of some) water level messages are received and the *RiskList* is computed, the EMS leaves the *risk_evaluator* role to enter the *riskmsg_sender* role (see LCC code below). During this simple role, an alarm message is sent to the emergency coordinator if the risk list is not empty.

```
a(riskmsg_sender(RiskList),EMS)::

 null <- RiskList=[]

 or

 alarm(RiskList) => a(emergency_coordinator,EC)
```

The LCC code below shows the *stop_sender* role taken by the EMS before the interaction terminates. Here, a stop message is sent to all sensors.

```
a(stop_sender(SPL),EMS) ::

 null <- SPL=[]

 or

 (
   null <- SPL=[H|T] then
    stop => a(sensor,H) then
     a(stop_sender(T),EMS)
 )
```

The role of a generic water level sensor is that of waiting for incoming requests. After a request (*water_level_request* message) is received, the constraint *getWaterLevel* is solved in order to retrieve the water level value *WaterLevel* sensed by the sensor. The message *water_level* is then sent to the requester (the EMS in our case). Among the already known parameters specified in such message, we find the *WaterLevel* and the *SensorName* parameters. The latest represents the identity of the data source, that is, the queried sensor.

```
a(sensor,S) ::

 (
  water_level_request(Timestamp,UnitMeasure) <= a(sender,EMS) then
   water_level(Timestamp,SensorName,UnitMeasure,WaterLevel)
    => a(receiver_risk_evaluator,EMS)
       <- getSensorName(SensorName) and
          getWaterLevel(Timestamp,SensorName,UnitMeasure,WaterLevel) then
   a(sensor,S)
 )

 or

 stop <= a(stop_sender,EMS)
```

In this interaction model, the role *emergency_coordinator* of the emergency coordinator (EC) is that of eventually receiving an alarm message from the EMS. At the reception, the EC takes the proper actions by solving the constraint *takeActions*. The actions taken will depend on the risk list passed as parameter in the message received. For example, if the risk registered in a given river bank is at the maximum level, the evacuation is started and the fire brigade is alerted. The role is recursive, so that the EC starts again to wait for alarm messages unless a stop message is received. The role described is shown below in terms of LCC code.

```
a(emergency_coordinator,EC)::

 (
  alarm(RiskList) <= a(riskmsg_sender,EMS) then
   null <- takeActions(RiskList) then
    a(emergency_coordinator,EC)
 )

 or

 stop <= a(emergency_monitoring_system1,EMS)
```

A-1.2 Weather_forecast.lcc

As anticipated before, this interaction model is used to periodically retrieve weather information from a weather forecast service. The peers participating to this interaction are the Emergency Monitoring System (EMS) and a weather forecast service. They play the main roles *weather_requester* and *weather_service* respectively. The EMS initiates the interaction by entering the *weather_requester* role. During this role the following parameters are set:

- *TotReqs*: the total number of requests that will be forwarded to the weather service. This parameter is set so to make the interaction terminating;
- *NoCurrReq*: the number of requests forwarded so far;
- *Location*: the geographical location about which the weather information are acquired;
- *WParams*: a list of weather parameters such as precipitation, temperature, humidity, etc;
- *NoDays*: number of desired forecast days;
- *IdleTime*: time elapsing between two consecutive requests.

Once the parameters are initiated, the EMS enters the role *weather_poller* which represents the core part of the EMS activity. Such activity consists on sending the message *weather_request* with the parameters *Location*, *NoDays* and *WParams* to the weather service. In our simulation, the location is Trento, the forecast is extended to 3 days and the weather information desired is the precipitation only. Once the above message is sent, the EMS receives as reply the *weather_conditions* message. The returned parameter *WParamsValues* contains the value of the weather parameters specified in the request. In our case, three values of precipitation are acquired, one for each day. After reception of the weather service reply, the EMS stores such values in its local database (*storeForecast* constraint) and stays idle for a time period equal to *IdleTime* before continuing the next request cycle (recursion

to the same *weather_poller* role). Notice that the cycle is repeated till the time when a number *TotReqs* of requests have been sent. After this time, a stop message is sent to the weather service and the interaction terminates.

```
r(weather_requester,initial)
r(weather_poller,auxiliary)
r(weather_service,necessary,1)


a(weather_requester,WR)::

 null <- setTotalWFRequests(TotReqs) and
         setNoCurrReq(NoCurrReq) and
         getLocation(Location) and
         getWeatherParams(WParams) and
         getForecastLenght(NoDays) and
         setIdleTime(IdleTime) then

   a(weather_poller(Location,WParams,NoDays,TotReqs,IdleTime,NoCurrReq),WR)


a(weather_poller(Location,WParams,NoDays,TotReqs,IdleTime,NoCurrReq),WP)::

 weather_request(Location,NoDays,WParams) => a(weather_service,WS) then

  weather_conditions(Location,NoDays,WParamsValues) <= a(weather_service,WF)

   then

     null <- storeForecast(Location,NoDays,WParamsValues) then

      (
        null <- sleep(IdleTime) and
                inc(NoCurrReq,NewNoCurrReq) and
                lessOrEqual(NewNoCurrReq,TotReqs) then

         a(weather_poller(Location,WParams,NoDays,TotReqs,IdleTime,
                          NewNoCurrReq),WP)
      )

      or

      stop => a(weather_service,WF)
```

The weather service takes the role *weather_service*. Here, it receives the request message *weather_request* from the EMS; then, it solves the constraint *getWeatherConditions* in order to get the output parameter *WParamsValues* from the input parameters contained in the request; finally, it returns such parameter back to the EMS (*weather_conditions* message) and recurses again to its role. If a stop message is received instead of a weather info request, the weather service ends its role.

```
a(weather_service,WS)::

 (
  weather_request(Location,NoDays,WParams) <= a(weather_poller,WP) then

   weather_conditions(Location,NoDays,WParamsValues) => a(weather_poller,WP)
    <- getWeatherConditions(Location,NoDays,WParams,WParamsValues) then

     a(weather_service,WS)
 )

 or

 stop <= a(weather_poller,WP)
```

A-2 Evacuation Interaction Models

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 "//".

A-2.1 Interaction Models used by simulator peers

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

A-2.1.1 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 inititated 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 SubSmulator_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 counted[20]. 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)


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 (*getNumbConnectedPeers* role) and updates the timeout *NewTimeout*;

[20]Notice 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.

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 sometimes 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 firefighter 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)
 )
```

A-2.1.2 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 A-2.1.4) 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)
  )
```

A-2.1.3 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*. Afther 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*[21], 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.

[21]Notice that this parameter is set in the "Simulation_cycles" interaction model which started first and is running in parallel.

```
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 receives two types of messages from the connected peer: *exit* and *await_ decision*. If the first message is received, the controller performs the costraint *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)
 )
```

A-2.1.4 Flood.lcc

This interaction model is used by the controller at every timestep, 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 A-2.1.1, 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 *updateFloodChanges(Changes)* constraint. The Java method implementing such constraint invokes a Prolog query[22].

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 (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:

```
nodeFloodLevl(nodeid, levl)
```

where `nodeid` is the identifier of the node received in the topology file at initialization time and `levl` 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:

[22]Some basic code of the controller peer is left in Prolog

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

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

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

A-2.1.5 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 A-2.1.1. 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*[23].: 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.

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

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)


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)
```

A-2.1.6 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 A-2.1.1. 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 once[24]. 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 A-2.1.1:

```
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
            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.

[24]The 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 A-2.1.1.

```
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)
   )
```

A-2.1.7 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 simulation[25], 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 dealed 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.

[25]Though generic, the actions currently performed are the "move" actions only.

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

a(action_performer,P)::

 (
   action(RegisteredName,Action) => a(simulator,S)
      <- connected() and
         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

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

A-2.2 Interaction Models used by network peers

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

A-2.2.1 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*[26] 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 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.

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.

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-2.2.3 for more details).

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

```
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)
```

```
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)
   )
 )
```

```
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-2.2.2 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*[27]. If the peer is prevented to make even the first step, most probably an inaccurate signaling by part of the Civil Protection Unit (CPU) 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.

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

```
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

      (
       //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 CPU 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-2.2.2 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-2.2.3 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 Unit (CPU).

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 CPU is very simple and constists in receiving the message *path_info_request* and serving the request. Notice that, to serve many requests, the CPU 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 database[28] 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*.

A-2.2.4 Querier-Reporter.lcc

This interaction used to model the communication between the Civil Protection Unit (CPU) 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 CPU 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 CPU 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.

[28]The knowledge base of the CPU is filled with information gathered periodically from the reporter peers by means of the "Querier-Reporter" interaction model (see next section for more details).

```
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)
    )


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)
```

The *querier* role entails two sub-roles: the *sender* and the *receiver* role. The CPU first gets the current timestep *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 [19]. After this, the CPU 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 CPU 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 CPU 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 timestep *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.

```
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)
  )
 )
```

After having received the message, the CPU 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 CPU expires and not all the reporters have replied, the CPU terminates the interaction thus missing some water level data.

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 correpond 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)
```