On-line Appendix to the Paper "MOBIDIS: A Pervasive Architecture for Emergency Management"

Fabio D'Aprano Massimiliano de Leoni Fabio De Rosa Massimo Mecella

Università di Roma "La Sapienza"
Dipartimento di Informatica e Sistemistica - DIS (sede decentrata di Latina)
{deleoni,derosa,mecella}@dis.uniroma1.it

This appendix includes some additional details about the pervasive architecture and experimental results of the paper "MOBIDIS: A Pervasive Architecture for Emergency Management", Proc. of 4th International Workshop on the Distributed and Mobile Collaboration (DMC 2006).[1]

1. Introduction

We decided to emulate real settings in order to test our pervasive architecture. Emulation is different from simulation: a realistic environment is built up by emulator computers. Thus devices are real and aren't aware about the presence of emulator they behave and communicate as they were really deployed in a wide area. The software installed on devices (PDAs and laptops) is developed for real scenarios and does function in emulated settings without modifications in its program code.

As a matter of fact, each device is connected to an emulator server. The server holds a realistic scenario containing obstacles, buildings, walls, ruins and many other objects that can stay in areas affected by disasters. Emulator adds also in that map some nodes representing team members forming the MANET: each virtual node corresponds to one real device connected to the emulator. When the software on device sends messages to another real one, such messages are actually sent to emulator that plays gateway rules. If nodes are in radio-range, according to the emulated scenario, emulator forwards messages to destination. Conversely if they are not directly connected, messages are dropped.

Indeed, the nodes in emulated area can move towards a destination (a tower, a street, etc) and their movements are not done casually but according to specific models.

We adopted the network simulator NS-2 [1] extended by Magdeburg patch for wireless emulation [2] for developing emulation server. NS-2 is born for batch emulation: the moments when events are raised, are decided before emulation starts. Movements are the typical events. The specification for a node to move towards a given destination has to be set up at design-time of the emulation.

Batch emulation, anyway, doesn't fit our needs. Events can not be determined at design-time in our approach. For example, if a given node in a given moment has to move towards a destination, depends on workflow scheduling and task assignment. Scheduling and task assignments depend on how the process carries on at run-time (as in real situations). So we wrote new event scheduler in NS-2, able to handle events which were not predicted before emulation started. The event scheduler is a module used by NS-2 that manages events such as network nodes movements, messages dispatching, etc. according to a given time plan.

Moreover, as NS-2 does not support any integration with external software, we wrote a TCL[2] TCP/IP server. This server enables an external client software to be able to interact with NS-2 on standard socket. In this way, the client software can set positions of emulated nodes. Also, it can invoke at run-time some commands in order to instruct NS-2 to move the virtual node in the emulated scenario or to get node positions.

The Figure 1 summarizes how real devices are connected to the emulator server and how they are mapped to nodes in the emulated area, to get extremely realistic experiments. For technical reasons, each real device has to be connected to NS-2 through a different wire-

[1] **Università di Roma "La Sapienza" Technical report 04-2006**

[2] TCL is the scripting language which NS-2 uses to configure emulations and simulations.

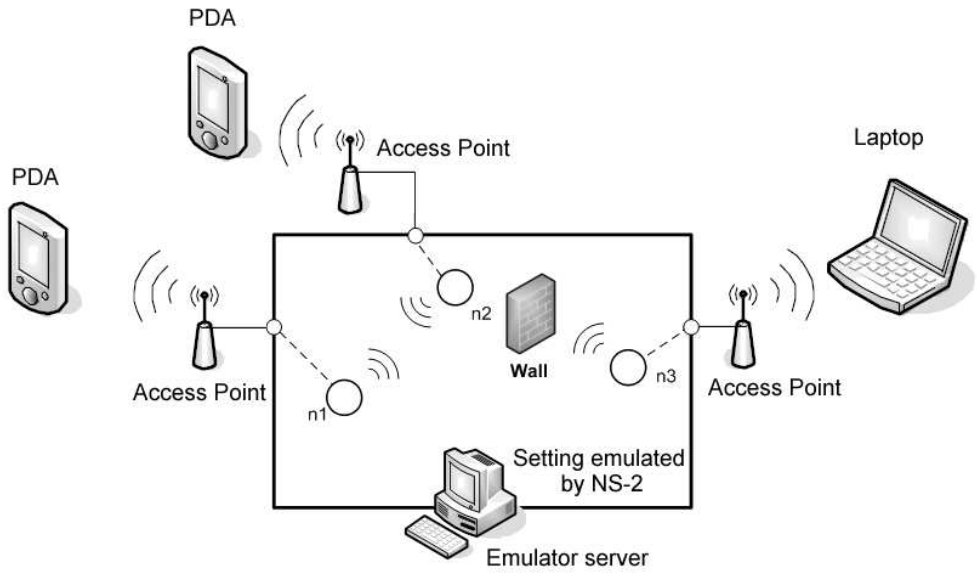

Figure 1. The hardware connection of the physical devices in experimental settings.

less or wired card of the emulator server.

As soon as possible, we will publicly release our modified NS-2 version under open source license.

2. Experimental Architecture

The TCP/IP server implemented as TCL Script in NS-2 is general and low-level. Indeed, it offers only very simple commands, such as `getPosition(idNode)`, `setDestination(idNode,x,y,speed)` or `setPosition(idNode,x,y)` where `idNode` is a integer number and `(x,y)` represents the coordinates of the destination (in the relative coordinates of the emulated environment). We have implemented in Java, on top, another server in Figure 2 named `Octopus Server`, able to handle obstacles and generic objects and by them, to let nodes to move according to more complicated models, such as Voronoi [3].[3]. The `Octopus Server`'s services are exposed both as a Java interface and TCP/IP server. The first one is used by the Graphic User Interface we implemented for a more friendly interaction with the Octopus Server; the latter one is used by actors. Figure 4 shows a screenshot of the Octopus GUI, during emulation. The position of nodes is the real one in such an emulation; blue arcs connect each pair of node in radio-range. Gray rectangles and lines represent, respectively, obstacles and Voronoi's paths. The former ones are put at design-time, the latter ones are directly computed before emulation starts.

The `NSI` lies at the bottom of the Coordinator and the generic devices (see Figure 2). `NSI` stands for *Network Service Interface* and it is a module installed on every mobile device in order to provide basic "send" and "receive" multi-hop calls over a MANET. This module acts as if it would send and receive messages to other devices (as in a real MANET). Although as a matter of fact, everything is sent to the emulator that decides if messages have to be forwarded to destination. `NSI` will go on functioning even when emulator is taken out.

NSI implementation is the same one of [5] that we developed and deployed for MANETs of Windows Mobile devices.

A generic Device defines a `NeighInfo` module which provides an interface for knowing which are neighbors in the MANET and the distances from them. Over emulation this information is held by the Octopus Server and, therefore, this module contacts the emulator to obtain that information by using the `Octopus Client` module. Octopus Client contacts its server counterpart through a TCP/IP socket. Octopus Client will not be needed any more, and only the NeighInfo module will have to be coded again. This one will invoke the software module provided by the specific hardware but the software interface to the other Generic Device modules

[3]Indeed, the Voronoi paths are decomposed as sequence of setDestination(x_i,y_i) where x_i,y_i are the coordinates of vertices of segments forming those paths.

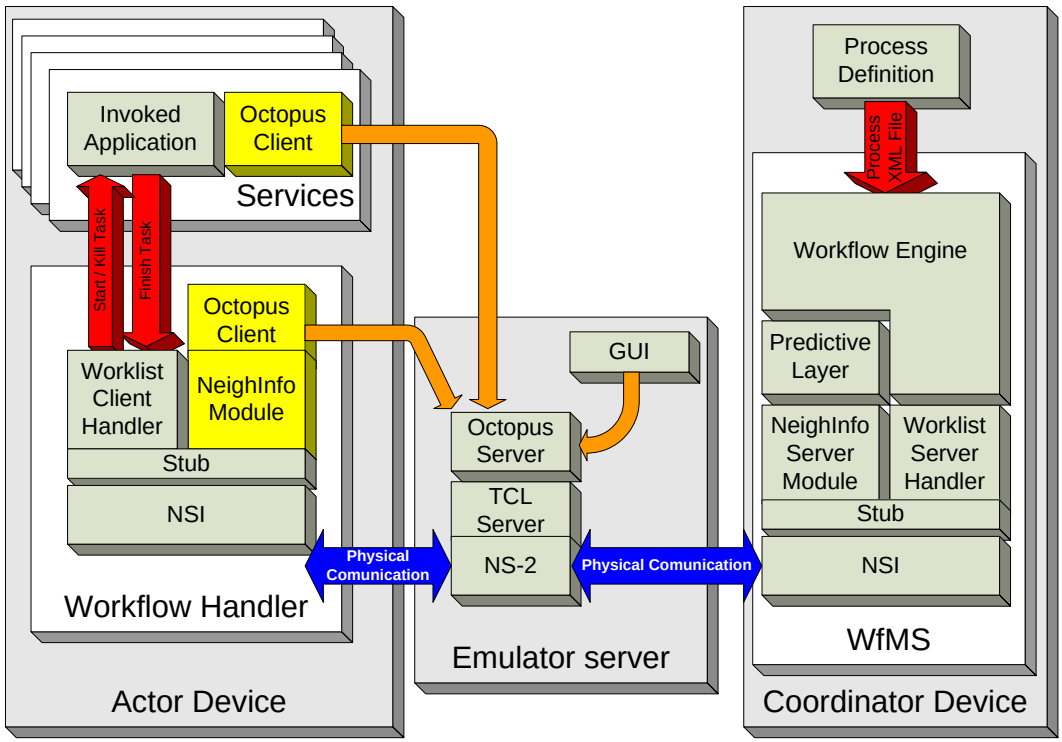

Figure 2. The Software modules composing effective architecture

will be unchanged. The `Worklist Client Handler`'s purpose is to contact the Coordinator in order to register, to obtain the list of available tasks to be performed, to pick a work-item and to inform the Coordinator about the end of task execution.

When actor picks one item from the worklist, the Worklist Client Handler gets information needed for the software able to carry on the corresponding task. Such a software is installed on the same device of the actor picking the workitem and represents a service provided by the actor. Possibly, such a software needs a particulary hardware which has to be present on the actor's device. The Figure 3 shows a possible software providing a service.

The Worklist Client Handler executes the software in separate process. When the client ends activity performance, he/she quits the corresponding software. The Worklist Client Handler detects this event and considers completed the task. So the information about task completion is sent to the Coordinator via NSI. If the task has to be aborted, Worklist Client Handle kills the process of the software. When software starts, it has to inform Octopus Server about the need for a movement (emulation scenario in which movement may be needed).

The coordinator contains a `NeighInfo Server` module upon the stub. This module is performed in a separated thread; it continuously asks to the actors for neighbor distances and it holds this information. The request is sent by NSI and on the devices it is handled by NeighInfo Module. This is needed for `Predictive Layer` Module which implements the predictive technique [4]; this module is implemented as a timer thread: at regular intervals a novel prediction is done. If a device is predicted to be going to disconnect, an asynchronous event is thrown to the `Workflow Engine`. It will have to look for a bridge and to push a support task for it. As a matter of fact, the pushing of tasks is not done directly by the Workflow Engine but it is forwarded to the `Worklist Server Handler`. This one will go to force corresponding work item to devices on behalf of the engine. The `Workflow Engine`, besides handling only disconnections, it takes care of managing process routing: when new tasks become enabled, the Worklist Server Handler is informed. The latter one creates corresponding item to be put in the Worklist. Moreover, Worklist Server Handler manages the worklist requests, the registrations, the task pickings and the signallings of task completion.

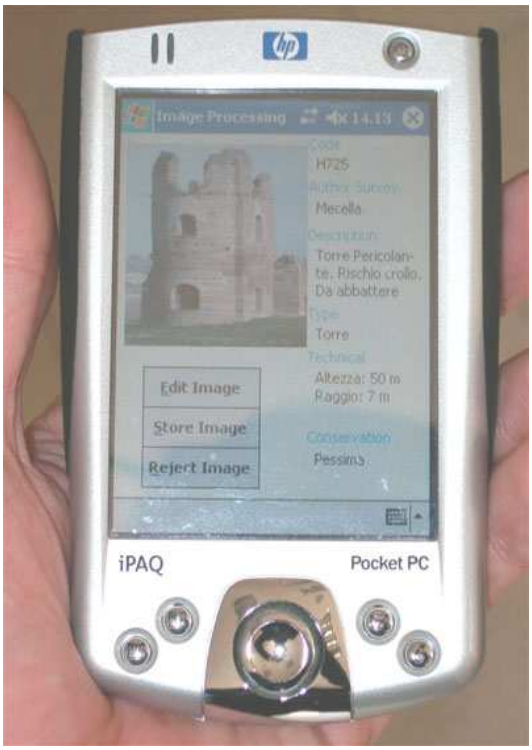

Figure 3. A possible software installed on a device to provide a service

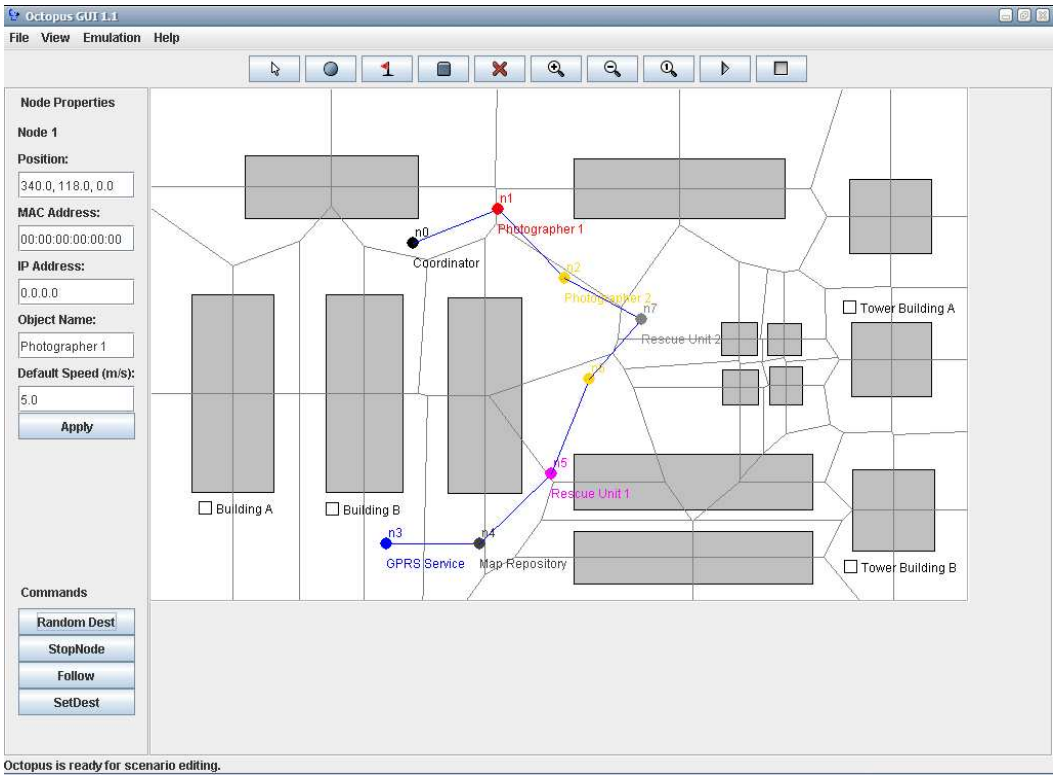

Figure 4. A Screen Shot of the Octopus Server GUI

3. Experiments Tuning

The Experimental architecture[4] described in the previous section is used to test our algorithms.

Experiments in this context are influenced by the initial positions of nodes and objects, and by the graphs of the processes. In our preliminary experiments, nodes, obstacles and other objects were manually put in the map at design-time by using the GUI of the emulation system, and process schemas were chosen with both loops and AND/OR splits.

The purpose of the first part of experiments is to tune some parameters of the algorithms. Once parameters has been tuned, we have performed more deep experiments.

The first tuned parameter is the *polling time*, i.e., the shortest time between two corrective actions; an higher value means more reactivity in doing corrective actions. The second parameter is β, i.e., the fraction of the radio-range the predictive technique doesn't signal a disconnection anomaly. As an example, in IEEE 802.11 with 100 meters of radio-range, β equal to 0.3 means that for a communication distance of 70 meters the prediction algorithm signals a probable disconnection.

β	**0.3**	**0.5**	**0.7**
polling time 3 sec	1%	0.09%	0.02%
polling time 5 sec	32%	4%	0,88%

Table 1. Experimental results.

The choices for parameter tuning are depicted in Table 1, varying polling time between 3 and 5 seconds and β between 0.3 and 0.7.

Specific experiments on the prediction techniques have been presented in [4]. Those experiments were based on simulated settings, so no prototypal architecture was implemented to be tested: the only purpose was to evaluate the prediction goodness.

4. Preliminary Results

A first set of experiments concerns how many predicted disconnections are resolved in order to evaluate the effectiveness of the bridging algorithm. The result is depicted in Figure 5, where the total number of disconnections in all experiments is shown. The first interesting result involves the number of disconnections predicted, effettive and resolved by bridging:

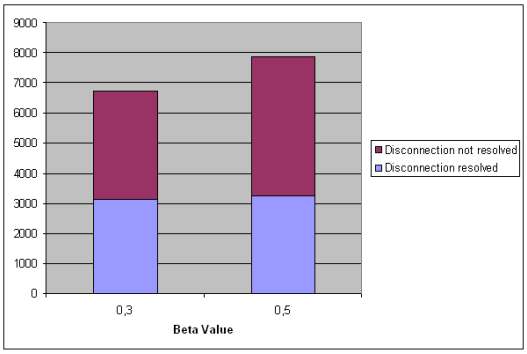

Figure 5. The total number of resolved and non-resolved disconnection in experiments

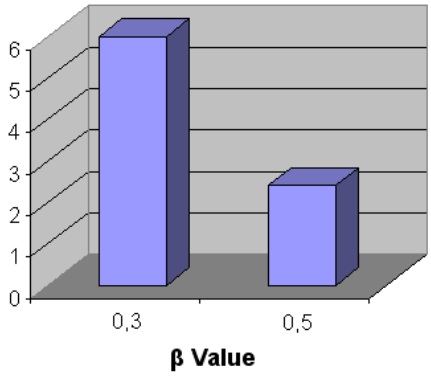

Figure 6. Connected components going to create in experiments

[4]It is possible to download a video-demo of the whole system at the URL `http://www.dis.uniroma1.it/~deleoni/documents/DMCVideo.avi`

it is invariant with respect to the chosen values for beta. Anyway, this preliminary result shows that just under half of disconnections are handled correctly and this is a good result. In fact, each disconnection means a roll-back action; avoiding half disconnection means the productivity is more or less doubled.

A second set of experiments analyzes the number of average connected components created during emulations. The result is depicted in Figure 6. Of course, the best situation is one connected components, that is no nodes goes ever out of range. In order to better analyze these data, consider that if new connected component is going to create during an experiments but after network becomes full connected again, then two connected components are considered to be created. After, if another connected component is born, then we consider three components to have been created in the whole experiment. Moreover, for $\beta = 0.5$ the mean value of created connected components is just over 2; that implies, even if disconnected MANET components were formed by four or five nodes, less than 10 nodes goes out of MANET range in a whole process performance.

These results are only a very preliminary validation of approach, to be refined in future works.

References

[1] The network simulator NS-2, `http://www.isi.edu/nsnam/ns`

[2] Daniel Mahrenholz, Svilen Ivanov, *Real-Time Network Emulation with ns-2*, University of Magdeburg, Germany.

[3] A. Jardosh, E.M. BeldingRoyer, K.C. Almeroth, S. Suri, "Towards Realistic Mobility Models For Mobile Ad hoc Networks", *Proceedings of MobiCom 2003.*

[4] F. De Rosa, A. Malizia, and M. Mecella. "Disconnection Prediction in Mobile Ad hoc Networks for Supporting Cooperative Work". IEEE Pervasive Computing, Vol.4, N. 3, 2005.

[5] F. De Rosa, M. Mecella: Designing and Implementing a MANET Network Service Interface with Compact .NET on Pocket PC. Proc. *3rd International Conference on .NET Technologies (.NET 2005).*