Open Dynamic Distributed Service Architecture Design and Modelling

Ghislain Putois, Philippe Bretier, Thierry Moudenc

***Abstract*—This paper presents a new methodology for the control and design of distributed service architectures in an open environment. A domotic service illustrates and introduces the modelling keypoints. In particular, an explicit modelling of the attentional mechanism is used to overcome the lack of a global state in distributed systems and the relative impossibility to explicitly model all external events.**

***Index Terms*—distributed architecture, design, control, open environment, attention**

INTRODUCTION

Distributed architectures are playing an increasing role in computer sciences to support open environment appliances, and especially in ambiant systems. But their complexity induces large design and validation costs, which justify the use of formal methods. Among these, the process flow modelling is a very sensitive topic. It guarantees the system nominal functionning and enables a good understanding of the supervised system. Moreover, the good design of such distributed systems is deeply rooted in their organic and processing flow structures, and should therefore rely on carefull design principles.

The traditional process modellings used in more commonly designed closed systems are based on state diagrams. However, none of them can properly address the open environment system needs, as they are all faced with a common major weakness: they all make the hypothesis of an explicit knowledge of a system global state and of the complete set of all possible system transitions.

Moreover, proposing a formal model for dynamic and distributed service architectures remains a daunting task. In particular, even though a must in open systems, mixing goal and data-driven approaches remains difficult with traditional modellings.

This situation has been concretised into a sample application, which illustrates the theoretical model we propose to overcome these issues.

Section I introduces the current state and directions regarding the design and control of a distributed system, as well as the major issue of global state. Section II details the additionnal difficulties faced when dealing with an open distributed system. Section III presents a sample application which illustrates all the major kinds of interactions present in a domotic system. Section IV lays a theoretical frame to address the mixing issue. Section V reveals how we propose to solve the control problem by explicitly modelling the attentional focus, and language concepts introduced to operate with our modelling.

I. TRADITIONAL APPROACHES AND MAJOR ISSUE

Traditional process modelling methods were designed to handle closed-environnment systems. Their common principle lies in representing all the states a system may be in, and all the possible transitions from a state to another, with their associated transition conditions. Well-known methods are Petri nets [1], the GRAFCET method [2], Hoare CSP [3] and Harel state diagrams [4]. Since, multiple variants have emerged for each of the cited methods, but they are all based on a common assumption that the system can be programmed in an imperative mode. The imperative paradigm postulates that a program can be described in terms of a sequence of actions.

However, in a distributed environment, there is no direct notion of global time, and a system cannot therefore be explicitly modelled by sequences of facts based on this global time. One can only define at best local informationally-consistent zones, and thus logical times [5] [6] and local sequences. However, the consistence of such logical times is dependent on the existence and reliability of communication channels between each part of the system, and their existence is possible only in a system where all participating parts are known by advance, which is unfortunately not the case in open environment. One must accept that sequentiality is now not the rule it was in monolithic systems, but rather an exception, or better an *ad-hoc* and crafted construct.

On an industrial point of view, current modelling approaches follow two different major directions, dependent on the research communities carrying them. The first approach is supported by the multi-agent community, which focuses on building communication protocols to enable transactions [7], but have neglected the more macroscopic design aspects of a large system. The second approach is supported by the robotics community, which focuses on implementing a reflex control based on a proprioceptive loop [8] [9], unfortunately inadequate for planning interactions.

The designer of a complex open system is finally left missing a methodology which would both enable macroscopic organisation and planning capabilities. In addition to the major show-stopper issue raised in the previous section, distributed open dynamic systems are also hard to comprehend due to their less well-defined forms. Nearly each of their aspects carry their lot of design problems.

II. ADDITIONAL DIFFICULTIES

While monolithic architectures are built upon a processor and a memory working together to run programs, distributed architectures multiply these materials in several distinct processing and data storage spaces, working autonomously, and which need to communicate with each other to run the programs through communication networks.

The communications are a potentially important point of failure of our distributed systems, as one cannot guarantee that a message sent by a device (*i.e.* a system part) will be instantaneously and correctly propagated and processed to another device of the system. Moreover, each device of the system may need to access the same data space or modify its content. One need to be very carefull when designing access control mechanisms. The control mechanisms is only a part of a more global problem of our distributed system, the synchronisation problem. Our devices need signalling protocols between processing spaces to enable the planning of more complex services. For more details on classical problems encountered by distributed systems, one can refer to [10].

As if distributed systems were not complicated enough, we also require our system to be dynamic and open, which means that some parts of the systems can be added or removed over the lifetime of the service. The transient nature of processing, data storage and communication spaces hinders the designer to anticipate all the possible configurations the system will take during its lifetime. Such condition make the communication capability very unreliable, since a message from a processing space might never be delivered to its

Authors are with Orange Labs, Lannion. France.
E-mails: firstname.name@orange-ftgroup.com

ROBOCOMM 2009, 31st Mar – 2nd Apr 2009, Odense, Denmark.
 ISBN 978-963-9799-51-6
DOI 10.4108/ICST.ROBOCOMM2009.5809

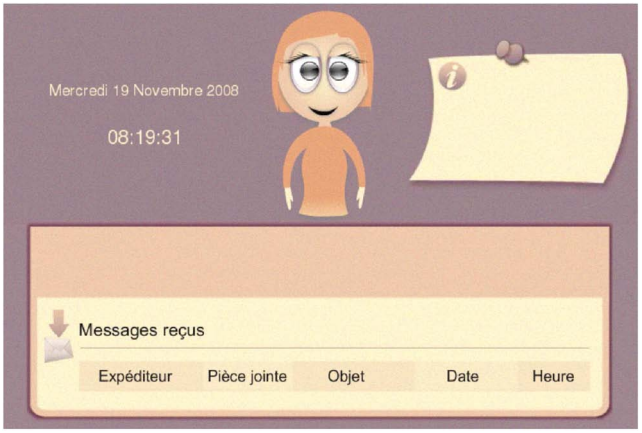

Fig. 1. Demonstration application: a graphical avatar presents the mailbox content

target, as both the target processing space and the communication space could disappear during the message transport time.

The low-level system aspects left apart, our service architectures are also faced with logical hurdles: we would like to build collaborative services by composition of elementary services offered by all the devices present at a given time in the system, and be able to dynamically add or remove new services in the system. In such conditions, a dynamic service cannot assume the set of services available on the system, nor how it will have to handle all the concurrency issues.

The previously raised issues highlight that the system has to manage two main sources of uncertainty: on a low-level layer, where the devices may appear and disappear, and where the communication between the devices cannot be properly guaranteed ; on a higher-level layer, where the services have to coexist and collaborate inside the system.

III. Application

We describe in this section a sample application developped which illustrates the modelling we propose to tackle the reviously raised issues. The system consists of a domestic simulation, composed of two vocal dialogue tasks (mailbox and address book) and two monitoring tasks (entrance captor and fire alarm), mediated by a graphical and vocal avatar.

The tasks implemented in the system have been chosen to illustrate the different strategies commonly used in dialogue:

- a user-directed task (vocal mail and address book),
- a non-prioritary interrupting task (the entrance detection),
- a prioritary interrupting task (the fire alarm).

The user-directed vocal mail task exhibits a good behaviour example where local sequentiality is needed: the user is guided through a set of phases to specify which mail she would like to consult. In a first time, the mails are sorted by their sender's name, and she user can ask for the mail of a specific user. If several users matches the request, an additional phase of dialogue is used to select the wished mail. The address book is organised similarly. Both dialogue tasks can be interrupted at any moment by a prioritary task like the fire alarm, which displays its own sequentiality. The dialogue tasks can also be disturbed by the non prioritary interrupting task from the entrance bell, which can intertwin a dialogue request in the middle of the current dialogue task.

The existence of interrupting tasks implies that a good design for such a multi-service system should not expect a sequence of behaviours to be atomically executed. On the contrary, the designer should try to limit the sequences of actions to the minimum as they are more than likely to be broken by unexpected events.

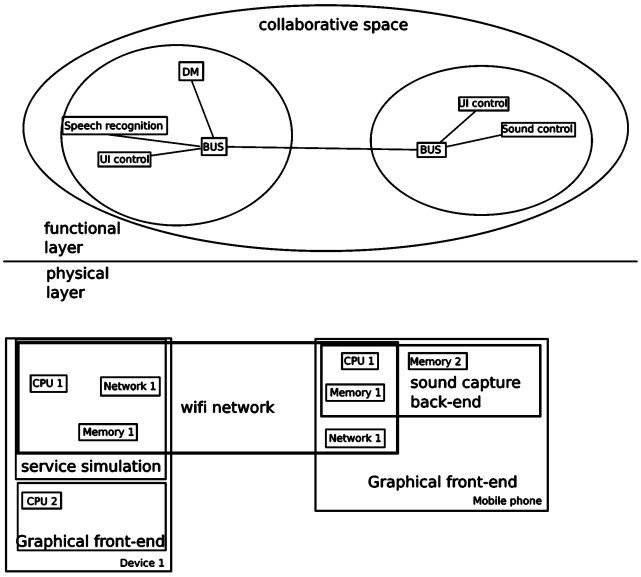

Fig. 2. Physical and functional layers: The physical layer represents the computer, the mobile phone and their applications. The functional layer represents how each abstract function (*a module*) sees each other.

IV. Layers of our modelling

This section details how we have modelled and organised our application. We propose a modelling of the distributed system into three layers to facilitate the design of services. The first layer describes the hardware components and their physical organisation. The second layer is an abstraction layer which structures the system functions and helps to define the action range of a service. Last, the third layer is the service design layer, which we use to model the service logic.

The figure 2 presents the first two layers. The third layer will be detailled in the section V.

A. The physical layer

Several views are relevant when considering a distributed system. One can naturally view the system as a set of physical devices. But one could focus on which parts of a device are able to work together, which defines an execution space. One could also consider the set of device parts able to work together across network connections, which defines a communication space.

Note that an execution space contains at least one processor, and a communication space contains at least two network interfaces or a shared memory.

In our application, the system is composed of two main devices: a mobile phone responsible for speech capture, and a computer which does the speech recognition, and manage the graphical user interaction. A wifi network binds both devices. The system also contains a presence detector and a fire detector.

B. The functional layer

In a distributed system, devices may appear and disappear at any time. A service should not rely on the existence of a specific device for all its life cycle. We introduce a loose binding between service functional needs and devices with a first abstraction called a module.

A module is an abstract entity that renders a set of functions. We see our whole distributed system as a structured network of modules, which changes over time.

All modules on a same execution space are able to work together, as are all modules on a same communication space. But there is more:

Digital Object Identifier: 10.4108/ICST.ROBOCOMM2009.5809
http://dx.doi.org/10.4108/ICST.ROBOCOMM2009.5809

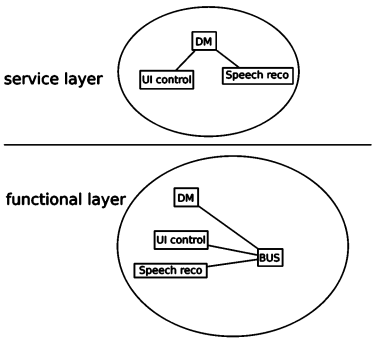

Fig. 3. Functional and service layers: even if each module are in reality linked to the bus, the service layer offers a hierarchical view to structure the service.

one can use overlap between an execution space and a communication space to define a larger space where all modules can cooperate. We define thus a collaborative space.

The collaborative space is the only informationally-consistent zone in a distributed system. In a collaborative space, one can properly define a coherent time, and solve synchronisation and concurrent data access issues. The collaborative space also defines the context each of its modules can leverage. It offers a consistent view on a functional part of a distributed system, and the problems of data sharing and synchronisation between modules can be well described in its scope.

In a collaborative space, there are two kinds of modules of particular interest: the bus message and the dialogue manager. The bus message is the functional module which enables the merge of an execution space and a communication space. It renders the message passing function for all modules inside the collaborative space. The dialogue manager is a module used to control the processing flow, and is the main topic of section V.

In our application, the functional layer consists of a lot of modules: it includes a sound capture module, a mobile phone front-end to enable the sound capture, a sound recogniser module, a semantic analiser module, some domestic interface modules (an OSGI gateway), a graphical avatar module, a text to speech module, a mailbox module, an address book module, a presence detector module, and a fire alarm module... as well as some logic controllers named Dialogue Managers.

C. The service layer

Inside a collaborative zone, a set of modules cooperate to realise a service by exchanging event message flows. The dialogue manager renders the processing flow control for a service. In figure 3, the functional layer represents how modules are really linked together, and the service layer represents how a service see the module organisation. All messages are routed through the Dialogue Manager module, which will maintain the service state consistency and realise its execution logic.

We describe in figure 3 a part of the controler embedded in our mobile phone: the UI control and the Speech Recogniser modules are subordinated to a Dialogue Manager, which manages the local dialogue logic to activate the sound capture.

V. Dialogue logic

We propose an original approach to define the control logic needed to organise the message flows, called thereafter the dialogue logic. The dialogue logic is a control logic that explicitly models the attention mechanism, which enables one to mix both goal and data-driven approaches inside the same representation.

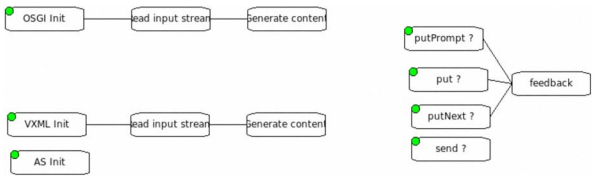

Fig. 4. A partial activation diagram: initialisation of some server modules on the computer on the left, and the feedback loop on the right.

A. States, attention and events

Traditional approaches are grounded on modelling containing states, transitions and guards.

In an open environment, any kind of event might occur, and no one can reasonably predict and design a complete state diagram. The asynchronous nature of the system prevents a good prediction of the next sequence of events, and makes an exhaustive data-driven state diagram unsuitable. We are however used to planning our actions step-by-step to fullfil a goal. We need a means to combine data-driven reaction and sequential goal-driven actions.

A way to successfully accomplish this is to understand that causality is not a prime phenomenon. It is rather a high-level construct which helps us to predict the events occuring in our environnment, after they have been processed by lower-level cognitive functions, as proposed by Broadbent in [11]. Those functions act as attention filters which sort out the relevant inputs from the large data event flow according to our current goal-driven expectations. Expectation is indeed the key to structure a data flow, and is explicitly model in our dialogue logic.

Our dialogue logic therefore models two kinds of objects: the goal-driven expectation state for given action plans, and the data-driven activation state for each expected state.

B. Distributed expectation state

Our modelling does not represent directly the current state of the system, but a set of "expected" future states. This modelling enables a more dynamic apporach. One can compare our system modelling to the attention handling phenomenon: we model the potentially interesting situations that our system expect to meet, as well as the behaviours it should adopt if it really meets these situations.

We define a state as a triplet of activation/situation/behaviour.

A state is said activable when it can match an expected situation ; it is said inactive when it does not. A state is said active once it has just matched its associated situation.

To render the sequentiality of a service, we bind the states with oriented arcs to indicate the expectation transitivity: once a state is realised, we made the bound states activable, so the system is now attentive to the situations which would trigger the bound states as seen in figure 4. In this case, the speech recogniser module and a VXML module are initialised, then the requests coming from the mailbox or address book modules will be sent as feedback to the graphical avatar module.

At any given time, the system's attention is composed of the states it has just activated, but also to the states previously activated. This set of active states models the set of relevant states in the near future.

C. Events, situations and behaviours

Each activable state can be effectively activated when the system or its environnement have changed, which is carried on by the reception of an event.

Every time an event is received, the system goes through the set of activable states and compares the current situation with the state-associated situation. If it matches, the activable state becomes activated.

Digital Object Identifier: 10.4108/ICST.ROBOCOMM2009.5809
http://dx.doi.org/10.4108/ICST.ROBOCOMM2009.5809

The system then applies all the behaviours corresponding to the set of active states. A behaviour is defined as a sequence of actions, where an action is the sending of an event, or a change of activation of any state.

D. Ilustrative usecase

We now introduce a simple usecase to ease the comprehension of the situation and action languages and logic algorithm we have implemented in our sample application.

The user is looking for a phone number in her address book, when something goes wrong in the kitchen, and a fire alarm is raised. The system then interrupts the dialogue task with the user to forcefully inform her that she should try to put out the fire. She tries and succeeds, and can resume her address book query.

In the system point of view, the user was going to ask for a contact's name in her address book. The system was therefore having its attention focused on recognising the name of the contact which the user would utter, but was also attentive on the domestic tasks: a part was attentive to the entrance bell, and a part was attentive to the fire detector. And the fire detector fused, so the task of sending the contact details is interrupted by the prioritary alert message. The system is now only attentive to the fire task, and temporarily disactivate all other tasks. Once the user has successfully put off the fire, the system becomes attentive to the paused task, and can present the requested information.

E. The situation description language

We need a situation description language able to express queries both on current service states and on event properties. We also need to be able to make complex requests by composing several simpler queries with logical operators.

The situation language we have defined is based on a simple object syntax:

- object: message.source, message.target, message.action, message.property("a_property"),
- context: dialogue.property("a_property"),
- binary operators: $=$, $>=$, $<=$, $>$, $>$,
- logical composition operators: *and*, *or*, *not*

In our usecase, when the fire detector has not yet triggerd, three action blocks are attentive: the first action block represents the address book dialogue, the second action block represents the entrance bell detector, and the last action block represents the fire detector.

```
fire_detector:
    (message.source="fire detector")
and (message.action="fire_begins")

entrance bell:
    (message.source="entrance bell")
and (message.action="bell rings")

address book:
    (message.source="speech recognition")
and (message.action="contact")
```

The situation should precise both the context of the event, and its main topic, as an event might often occur in different context. For instance, the recogniser module will fire its detection events in every dialogue situation, both for address book querying and when acknowledging the fire extinction.

F. The behaviour description language

We need a language both able to modify the current service state and to build and send event messages to the other modules used by the service.

We introduce the following language:

- service("module","action") to invoke a module method by sending an event,
- parameter("name","value") to create the event context,
- importParameters to copy the context of the received message,
- activate(block_id) and disactivate(block_id) to modify the service state activations,
- next to ease the making of behaviour sequences.

In our application, the behaviour called when a fire alarm is triggered would be:

```
paramater("Warning","Alert! Fire in the kitchen!")
service("avatar","displayAlarm")
service("tts","play_alarm")
disactivate(address_book_task)
disactivate(entrance_bell_task)
next
```

The graphical avatar module will be given the order to display its warning message, the text to speech module will at the same time be ordered to play a fire alarm, and the other non-prioritary tasks will be put out of attention.

G. Algorithm

The algorithm 1 presents the three phases used in the dialogue logic: first determine the attentive active states between all the attentive states, realise their actions, and finally compute the next attentive states.

Algorithm 1 Dialogue logic

while $E_{activables} \neq \emptyset$ **do**
  an *event* is received
  compute active states:
  $E_{actives} \leftarrow 0$
  for $e = (condition, behaviour)$ in $E_{activables}$ **do**
    if *event* matches *condition* **then**
      $E_{actives} \leftarrow E_{actives} \cup \{e\}$
    end if
  end for
  realise relevant actions:
  for $e = (condition, behaviour)$ in $E_{actives}$ **do**
    $E_{activables} \leftarrow E_{activables} \setminus \{e\}$
    execute *behaviour*
  end for
  propagate activabilities:
  for e *in* $E_{actives}$ **do**
    $E_{activables} \leftarrow E_{activables} \cup successors(e)$
  end for
end while

At the beginning of our usecase, the activable states are the first state from the address book dialogue, the frst state from the entrance bell monitoring, and the first state from the fire detection. When the fire event is received, the first fire detection state matches with the event and becomes actives. All other states remains activable. Then, the behaviour for this first fire detection state is realised ; the active set becomes empty, and the fire detection state behaviour is executed (as described in the previous subsection). Then the second fire detection state become activable.

Then, once the fire is extinguished, the "fire off" event is sent by the fire detector, and the second fire detection state is realised. This state reactivates the previously paused address book and entrance bell dialgoues, and reinitialised the first fire detector state, in case a second fire happens.

Digital Object Identifier: 10.4108/ICST.ROBOCOMM2009.5809
http://dx.doi.org/10.4108/ICST.ROBOCOMM2009.5809

CONCLUSION

The control logics are traditionally focusing on closed systems, where one can anticipate every future occuring events, and model a complete planning of the system's behaviour. They lay on the fundamental assumption that the system can be considered as a whole, and that every action can be modelled as a clear modification of a global system state.

Unfortunately, in an open environment, this fundamental assumption does not hold anymore, because the notion of a system global state does not exist. However, one can define some stable collaborative zones, in which one can define a consistent macroscopic state.

In the colllaborative zone, it remains difficult to anticipate every kind of events the zone can receive. An indirection level of planning has therefore been introduced to overcome this problem. This indirection level of planning is based on the modelling of an explicit attentional mechanism. Such a system has led to the implementation of a demonstrator in the field of domestic appliances.

The development of a dedicated middleware and the associated tools paves the way for new methodologies to structure, design and control complex systems.

REFERENCES

[1] C. A. Petri, "Kommunikation mit automaten," Ph.D. dissertation, Universität Bonn, Bonn, Germany, 1962.

[2] G. A. S. Logiques, "Pour une représentation normalisée du cahier des charges d'un automatisme logique," *RAII*, vol. 62, pp. 36–40, 1977.

[3] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, pp. 666–677, 1985.

[4] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987.

[5] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[6] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms*. North-Holland, 1989, pp. 215–226.

[7] FIPA, "Fipa communicative act library specification," Tech. Rep., December 2002.

[8] J.-C. Baillie, "Urbi: towards a universal robotic low-level programming language," *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pp. 820–825, Aug. 2005.

[9] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based rt-system development: Openrtm-aist." in *SIMPAR*, ser. Lecture Notes in Computer Science, S. Carpin, I. Noda, E. Pagello, M. Reggiani, and O. von Stryk, Eds., vol. 5325. Springer, 2008, pp. 87–98.

[10] T. Kindberg, J. Dollimore, and G. Coulouris, *Distributed Systems: Concepts and Design (4th Edition)*, 4th ed.

[11] D. Broadbent, *Perception and Communication*. Pergamon, 1975.

Digital Object Identifier: 10.4108/ICST.ROBOCOMM2009.5809
http://dx.doi.org/10.4108/ICST.ROBOCOMM2009.5809