

The Role of Coordination and the Inadequacy of Current Approaches

In the previous chapter, we have identified the need for novel approaches to software engineering, promoting adaptive self-organization, and context-awareness. Now, we start to analyze the important role that will be played by the identification of proper coordination models in the process toward the definition of a suitable innovative software engineering approach.

In general terms, a *coordination model* identifies the mechanisms and the policies according to which an ensemble of “actors” can orchestrate the overall activities. Such an orchestration include mechanisms and policies for both exchange of information and synchronization of activities. The study of coordination models goes beyond computer science [86], in that also behavioral sciences, social sciences, business management, and logistics somewhat strictly deal with how various types of actors (e.g., animals, humans, trucks) can properly coordinate with each other.

In this book, we obviously focus on the coordination of computational actors, i.e., the components of distributed applications and systems. For the sake of simplicity, we will often adopt the generic term “agent” to indicate any component of a distributed network scenario hosting a computational activity that needs to coordinate with other components. These will this include actual software agents, processes in a distributed application, peers in a P2P network, as well as mobile devices and computer-based sensors (or, which is the same, the system-level processes running over them). Such a generalized adoption of the agent term, though, is not arbitrary [165]. In fact, it refers to entities with some degree of autonomy in execution, interacting with other entities, and situated in some context/environment. That is, entities matching the canonical definition of agents provided by the agent community [162].

3.1 The Fundamental Role of Coordination Models and Infrastructure

Whatever the scenario of interest, agents acting in the context of distributed applications and systems have the primary need to interact and coordinate with each other to achieve their goals. In all the scenarios described in the previous chapter, coordination between the agents constituting the system has a fundamental role.

- In the **micro scale**, where each single agent (i.e., micro device) has limited power and resources, coordination is mainly required to let the agents cooperate to accomplish tasks that exceed their capabilities as single individuals. In the “pipe repairing” application (described in Subsect. 2.1.1), for example, no single agent is large enough to repair a hole in the pipe; they must cooperate to aggregate into a suitable structure to fix the hole. As another example, in almost all sensor network applications, coordination between single sensors is enforced to exceed the sensing power of each single device or to implement power saving policies. Sensors, in fact, by coordinating with each other, can average their measures to wash out environmental noise, or can coordinate duty cycles to save battery energy.
- In the **medium scale**, agents are the applications running on handheld, wearable, and embedded devices. Coordinating their activities is at the core of a lot of pervasive computing applications, where humans can take advantage of a proper orchestration of distributed activities to improve their interactions with the surrounding environment. Consider, for example, the case of a housewife who “asks” the kitchen for suggestions about what she could prepare for dinner. The answer could be provided by having a computer-based fridge analyze what food is in it, by having computers embedded in the kitchen’s shelves do the same, and by having them all exchange this information and cooperatively verify on an electronic recipe book what can be prepared on this basis. That is, answering to the housewife question implies a proper coordination of activities among the various pervasive computing devices in the kitchen.
- In the **global scale**, again, coordination is of primary importance. For example, in Internet-scale P2P applications, agents (i.e., peers) need to coordinate with each other in order to dynamically discover communication partners and to autonomously engage direct or third-party interaction patterns. For example, every kind of routing problem (whether on a real network [20] or on an overlay virtual network [120, 129]) can be easily regarded as a minimal coordination problem: autonomous nodes must cooperate, forwarding each other packets, to let the packets flow from sources to destinations.

The pervasiveness of coordination activities and their primary role in achieving global application goals in any distributed application and system,

clearly make the identification of proper coordination models of fundamental importance. Not surprisingly, the research area of coordination models and middleware is particularly crowded and attracts researches from different communities like agents [56], software engineering [58], and distributed computing systems [57].

Coordination activities may vary from simple mutual exclusion policies to access shared resources to complex distributed artificial intelligence algorithms for collective problem solving. But, whatever the case, the design and development of complex distributed applications always call for the identification of a coordination model facilitating the overall design and development process. In the case of the emerging scenarios of interest to this book, such a coordination model should be able to facilitate adaptive self-organization of activities, and should be complemented by proper middleware to support the execution of distributed applications.

In particular, we think that any coordination model and the associated supporting middleware should provide

1. suitable mechanisms to enable coordination, i.e., interaction and synchronization mechanisms;
2. suitable means to promote context-awareness;

With regard to the first point, coordination requires by definition some form of communication between agents and some form of synchronization of activities. Besides sharp means to enforce communication and synchronization (e.g., messages, semaphores, etc.) one should also account for less obvious means, i.e., indirect interactions mediated by an environment (also known as “stigmergy”) or behavioral interaction (i.e., indirect interactions induced by agents observing each other’s actions).

With regard to the second point, a coordination model for dynamic and open scenarios also requires some forms of context-awareness. In fact, any agent has to be somehow aware of “what is around,” i.e., its context, to meaningfully work in a specific operational environment, and to properly combine efforts with other agents. However, when agents are embedded in a possibly unknown, open, and dynamic environment (as in the case of the depicted emerging scenarios), they can hardly be provided with enough *a priori* up-to-date contextual knowledge, and should be supported in the process of dynamically acquiring it.

The above two points, interaction mechanisms and context-awareness, are indeed strictly intertwined, in that contextual information can be communicated only by the available interaction mechanisms. With this regard, it is worth anticipating that on the one hand, indirect interaction mechanisms appear much more suited for coordinating activities in open and dynamic scenarios, in which agents can appear and disappear at any time. In fact, these models uncouple the interacting entities and free them from the need for directly knowing each other to interact. This promotes spontaneous interaction, which is the basic ingredient to support self-organization. On the other hand,

for spontaneous indirect interactions to take place in an adaptive way, agents must somehow affect the surrounding environment by their actions in a way that can be somewhat perceived by other agents. That is, indirect interactions require the capability of affecting the context and of perceiving the context.

As a simple example, in cooperative distributed robotics, a lot of implementations rely on robots interacting by merely observing each other's actions. Robots can acquire a picture of the surroundings with their cameras and, using such a picture, decide what to do without any explicit communication being involved [67].

In addition, the characteristics of modern distributed scenarios analyzed in the previous section also alert us that any coordination model, for being effective, should also promote locality both in interactions and in the acquisition of contextual information. In fact, for systems which can be characterized by a large number of decentralized agents, any approach requiring global-scale interactions is doomed to fail. Scalability and ease of management can be properly supported only by a model in which most interactions occur at a local and localized scale. In the simple case of cooperative robotics, for instance, one cannot rely on the fact that each robot sees and understands the actions of all other robots: that could work only for small environments with full line of sight, and in any case the presence of a large number of robots would challenge the limited capabilities of robots.

3.2 An Exemplary Case Study Application

To exemplify and fix ideas on what has been discussed so far, it may be useful to introduce a case study application. The chosen application involves a pervasive computing scenario, and in particular a computer-enriched museum in which tourists, while visiting it, can exploit PDAs or smart phones to get a better and more immersive experience. A number of devices embedded in the museum can provide tourists with a sort of interactive guide, but they can also be exploited by museum guards for the sake of monitoring and control.

In particular, for tourists, the pervasive services provided by the museum infrastructure may be of help to retrieve information about art pieces, effectively orient themselves in the museum, and meet with each other (in the case of organized groups). For museum guards, the pervasive services can be used to improve their monitoring capabilities over art pieces and tourist actions, and to coordinate each other's actions and movements. In the following, we will concentrate on two specific representative problems: (i) how tourists can gather and exploit information related to an art piece they want to see; and (ii) how they can be supported in planning and coordinating their movements with other, possibly unknown, tourists (e.g., to avoid crowd or queues, or to meet together at a suitable location).

To this end, we assume that (i) tourists are provided with a software agent running on some wireless handheld device, like a PDA or a cell phone, giving

them information on art pieces and suggestions on how and where to move; (ii) the museum is provided with an adequate embedded computer network. In particular, embedded in the museum walls (associated either with each art item or each museum room), there are a number of computers capable of communicating with each other (by wired or wireless links) and with the mobile devices located in their proximity (e.g., by the use of short-range wireless links); and (iii) both the devices and the infrastructure hosts are provided with a localization mechanism to find out where they are actually located in the museum; this could be implemented by some kind of cheap mechanism relying on well-known algorithms based on radio or acoustic signal triangulation [50].

Despite this coarse description we think that this kind of case study captures in a powerful way features and constraints of next-generation application scenarios:

- It can be of very large size. In fact, in huge museums there can be thousands of embedded electronic devices and hundreds of tourists with mobile devices. There can be multiple systems concurrently running within the museum computer infrastructure (e.g., light and heating control systems) and other systems connected to these other services. In addition, since a huge museum can have multiple sections managed by different organizations, some degree of decentralization may also be present.
- It represents a very open and dynamic scenario. In fact, a variable number of unknown tourists may enter and leave the museum at any time, each following unpredictable schedules and visiting plans. In addition, the museum too can exhibit high dynamics, in that huge museums very often restructure their topology to host temporary special exhibitions, and very often art pieces are moved from room to room and new art pieces are added.
- The need for context-awareness is intrinsic in the goals to be pursued by tourists and museum guards when exploiting the infrastructure, in that they all somewhat relate in understanding what is happening in the museum and act accordingly.

The above characteristics carry on a number of implications. Despite the high dynamics of the scenario, the system should be robust and flexible. When embedded hosts break down, wireless networks have glitches, or other unexpected malfunctioning occurs, the system should exhibit a limited and gradual decay of performance. When special exhibitions take place or new art pieces are introduced, the system should immediately reflect the new configuration in the way it provides its services, and without any temporary unavailability. Whenever a tourist enters the museum, he must be immediately allowed to take advantage of the museum services.

From the viewpoint of the museum infrastructure, one cannot rely on manual configuration and reconfiguration for the above requirements to be

fulfilled. In fact, the human efforts required to do that would be nearly continuous and dramatically expensive. Also, for very large museums, the time required for such reconfigurations would lead either to temporary unavailability or to services providing obsolete information. The only feasible solution is to have the system be able to autonomously configure its operational parameters in response to changed conditions, in an adaptive and context-aware fashion, without requiring human intervention and without exhibiting perceivable service malfunctioning.

From the viewpoint of tourists, they must be properly enabled to access all the available information in an up-to-date way, and they must be given the possibility of understanding how to move in the museum. Clearly, this should be done on the fly for each tourist, by having his mobile device spontaneously connect with the embedded infrastructure, and by having the whole system dynamically provide personalized services (e.g., “Since you are interested in both Egyptian art and Greek sculpture, here the best path for you to follow based on current crowd conditions”). Also, tourists and museum guards should be enabled to dynamically coordinate with each other (e.g., “All students of my class please report to me!”). For this to occur, tourists and museum guards must be able to start interacting possibly without knowing each other a priori (e.g., “Anyone here interested in discussing Egyptian art?”) and in a context-aware way (e.g., “Alice, this is Bob, let’s walk toward each other to meet in between”). Again, since we cannot assume any possibility of centralized control, all these types of context-aware interactions must be promoted by the system in an adaptive way without requiring any manual configuration.

As an additional note, it is worth noting that even testing and debugging a system of that kind is extremely difficult. That would imply accounting for an uncountable number of possible situations (e.g., what is the typical group behavior of a class of children visiting the museum? What happens when someone shouts “fire!” in a packed room?). For this reason, testing should also follow a different approach. Rather than trying to account for all possible situations to verify that the system is flawless, one should structure the system so as to make it able to dynamically adapt itself to face any unexpected situation.

To be successful, any approach to designing and developing a system capable of exhibiting the above characteristics should rely on the choice of a proper coordination model and by a corresponding supporting middleware infrastructure. The model should promote spontaneous interactions among agents that possibly do not know each other a priori (e.g., a new art piece that connects to the museum infrastructure, or two tourists with common interests that want to meet to discuss with each other) and should enforce context-aware interactions in an expressive way, to ensure that any dynamic adaptation of the system and any coordination activity (e.g., a group of museum guards that wants to monitor in a coordinated way different areas of the museum) properly reflect the current conditions of the system and of the other agents in it.

In the rest of this chapter, we will refer to the above case study to evaluate the inadequacy of current coordination models and middleware to face the complexities of modern scenarios. Moreover, from time to time in the book, we will again revert to this case study to ground the discussion.

Although the case study focuses on pervasive computing (which also unveils our specific specific area of interest) it introduces issues which are of a very general nature. In fact, it is analogous to a number of different scenarios, e.g., traffic management and forklift activity in a warehouse, where navigator-equipped vehicles can guide their pilots on what to do; mobile robots and unmanned vehicles exploring an environment; spray computers having to organize their relative positions and activity patterns; software agents exploring the Web, where mobile software agents coordinate distributed researches on various Web sites. Therefore, all our considerations will be of a more general validity.

3.3 Inadequacy of Current Approaches in Supporting Coordination

Most coordination models and middleware used so far in the development of distributed applications appear, in our opinion, inadequate in supporting coordination activities in dynamic network scenarios, such as those described in the previous chapter and in the above case study. In the following paragraphs, we are going to survey various coordination models and middleware to illustrate their inadequacies from a software engineering perspective. The analysis will be mainly focused on evaluating how those models and middleware provide agents with contextual information and whether the information provided is suitable for supporting effective coordination activities.

We identified three main general classes of coordination models encompassing almost all the proposals. These include (i) direct coordination models, i.e., message passing and client-server ones; (ii) shared data space models, i.e., tuple space ones; and (iii) event-based models. The implementation of middleware infrastructures to support a specific model within a class can be very different from each other (e.g., centralized vs. distributed, or using proprietary vs. open protocols). Still, they are mostly equivalent from the software engineering viewpoint, in that the overall design of an application developed adopting a specific middleware would not be substantially affected by being ported on a different middleware relying on a coordination model of the same class.

3.3.1 Direct Coordination Models

Models based on direct coordination promote designing a distributed application by means of a group of agents that can coordinate by directly communicating with each other in a direct and explicit way, by message passing or in

a client-server way (i.e., adopting some kind of remote procedure call mechanism). Client-server middleware systems like Jini [65], and message-oriented middleware like UPnP [150] and JADE [10], are examples of middleware infrastructures rooted on a direct coordination model.

The model

Since, from a software engineering point of view, all the implementations of direct coordination models are rather similar, here, for simplicity, we will focus the description according to the terminology adopted by the Jini middleware (see Fig. 3.1). At the end of this section we will briefly review the main differences between client-server and message passing implementations.

The main service offered by direct communication models, like Jini, is lookup and discovery. The main idea at the bottom line of this service is to provide agents with a shared middleware in which they can store their identities and capabilities, and in which they can look for other identities and declared capabilities to find suitable interaction partners. In particular, this service can be implemented by means of either white or yellow pages.

- A *white page* server basically provides a database where agents can store their name together with the network address and port they are listening to. Agents connect to the server either to publish themselves on the network by storing their own identities, or to look for the address of agents with which they want to interact. After obtaining such information agents can communicate directly (i.e., through sockets) with each other. Although this service decouples the agent symbolic names from the host in which they are running, it actually requires an a priori (i.e., compile time) acquaintanceship between the agents.
- A *yellow page* complements white pages, by allowing an agent to associate a machine readable description of their capabilities with their network address. This allows a better decoupling of agent interaction, in that an agent can look for the specific service it needs, disregarding the identity of the agent providing that service.

In Jini, for example, a specific lookup and discovery server provides the above functionalities. To give agents access to the server it is possible to install it at a well-known network address, or the agents can start a local network broadcast search.

Once an agent connects to a newly discovered one, a communication problem arises: *the agents need to talk the same language*. Earlier proposals, like Jini, adopt a client-server approach: agents export an interface (in the object-oriented sense) and other agents can invoke methods on that interface disregarding the methods' actual implementation. More recent proposals like JADE [10] do not encode the agents' interaction by means of method invocation, but by means of formatted text messages. Agents receiving such messages must

be able to understand the message syntactic and semantic content to decide which action to undertake. Such understanding is typically promoted by the adoption of shared ontologies between agents [10].

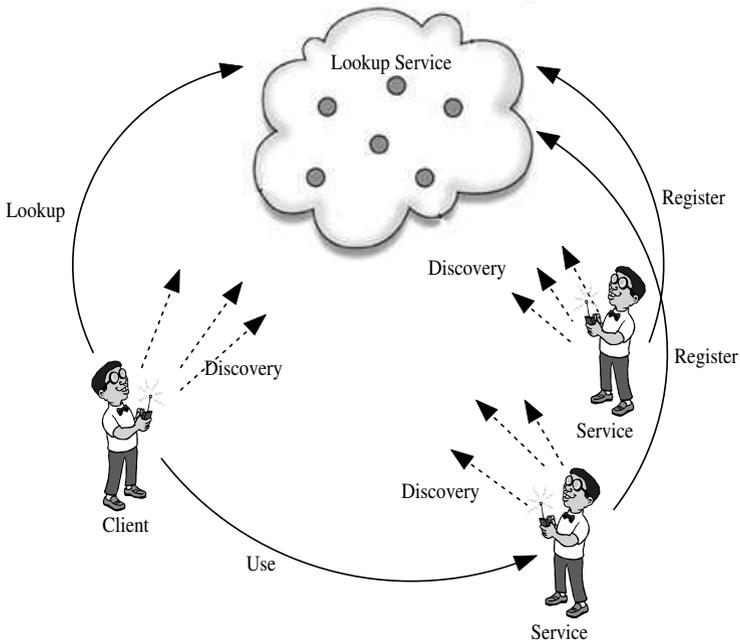


Fig. 3.1. Direct coordination: Jini main operations

Inadequacy

One problem of direct coordination approaches, as promoted by the adoption of the Jini middleware or of an alike middleware, is that agents have to interact directly with each other and can hardly sustain the openness and dynamics of near future computing scenarios. Firstly, explicit and expensive discovery of communication partners – as supported by directory services – has to be enforced for enabling partners that do not previously know each other to interact. Secondly, agents are typically placed in a “void” space: the model, per se, does not provide any contextual information: agents can only perceive and interact with (or request services from) other agents, without any higher-level contextual abstraction.

In the case study scenario, tourists have to explicitly discover locations of art pieces and of other tourists. Also, to orchestrate their movements, tourist must explicitly keep in touch with each other and agree on their respective

movements by direct negotiation. These activities require notable computational and communications efforts and typically end up with ad hoc solutions – brittle, inflexible, and nonadaptive – for a contingent coordination problem.

To better clarify these ideas, let us focus the attention on the meeting problem in the museum case study. Specifically, let us consider the case in which a group of agents wants to meet in the best room according to the current locations of the agents (i.e., at the center of gravity of their current positions). The pseudo-code in Fig. 3.2 implements an agent performing the meeting application by exploiting the services of a Jini-like middleware.

```

01: // register myself to the discovery middleware
02: middleware.register(this)
03: // get a reference to the other agents in the meeting group
04: for every name in the meeting group
05:   agent[i] = middleware.get(name)
06: end for
07: // get the museum map
08: museum = middleware.get(MuseumMap)
09: //proceed with the meeting
10: while not meet
11:   // find where the other agents are
12:   for every agent in agent[]
13:     location[i] = agent[i].getLocation()
14:   end for
15:   // compute the best room for the meeting on the basis
16:   // of the agent current locations and museum map
17:   room = computeBestRoom(museum, location[])
18:   // move toward the meeting room
19:   goTo(museum, room)
20: end while

```

Fig. 3.2. Pseudo code of the meeting application with a direct coordination middleware

Looking at the pseudo-code, the problems inherent in direct coordination models are immediately evident. First of all, the system relies on global middleware services that can be difficult to implement and can represent a bottleneck or a single point of failure. Secondly, the system does not cope gracefully with situations in which agents can dynamically join or leave the meeting group (in rows 4-6 and 12-14, the members of the meeting group are supposed to be fixed). Thirdly, a notable decision burden is left to the agents. Agents have to exchange information about their current positions and evaluate by themselves the best room for the meeting, by merging information about the museum map and other agent current locations (row 17). Moreover, they have to implement some navigation (i.e., routing) algorithm to move to

the destination room within the museum map (row 19). Whenever some contingency occurs or some new information is available (i.e., a room in the path of an agent is discovered to be so crowded that it should be best avoided), the agents have to explicitly renegotiate a new meeting point.

For all these reasons, direct coordination models are not suited to effectively support agent coordination activities in dynamic scenarios.

3.3.2 Shared Data Space Models

Coordination models based on shared data spaces support agent interactions with the mediation of localized shared data structures, which agents can read and write, and which could also be used for representing contextual information. These data structures can be hosted in some data space such as a tuple space, as in EventHeap [66], JavaSpaces [39], and TSpace [83], or they can be carried by agents themselves and dynamically merged with each other to enable interactions, as in Lime [112] or XMiddle [96].

The model

Let us refer to the tuple space model, the most general and widely used model based on shared data spaces.

A tuple space is a shared, associatively addressed, memory space, organized as a multiset, i.e., as a bag of tuples. The tuplespace concept was originally proposed in the context of the Linda coordination language [43], and has recently received renewed attention because of several innovative proposals, like Sun's JavaSpaces [39].

The basic element of a tuple space system is the tuple, which is simply a vector of typed values, or fields. Templates are used to associatively address tuples by pattern matching. A template (often called anti-tuple) is also tuple, but with some (zero or more) fields in the vector replaced by typed placeholders (with no value) called formal fields. A formal field in a template is said to match an actual tuple field if they have the same type, whatever the value in the actual field of the tuple. If the field of the template is not formal, both fields must also have the same value to match. Thus, a template matches a tuple if they have an equal number of fields, with types respectively corresponding, and each field of the template matches the corresponding tuple field.

A tuple is created by an agent and placed in a tuple space by a *write* primitive (called *in* in the original version of Linda). Tuples are read or extracted by a tuple space with *read* and *take* primitives respectively (the latter called *out* in the original version of Linda), which take a template and return the first matching tuple. Since a tuple space is an unstructured multiset, the choice among multiple matching tuples is arbitrary and implementation-dependent. Most tuple space implementations provide both blocking and non-blocking versions of the tuple retrieval primitives. A blocking read, for example, waits

until a matching tuple is found in the tuple space, whereas a non-blocking version will return a “tuple not found” value if no matching tuple is immediately available.

Tuple spaces provide a simple, yet powerful mechanism for agent coordination. Tuple space-based programs are easier to write and maintain, because tuple-based interactions uncouple interacting agents. Destination uncoupling is enforced because the creator of a tuple requires no knowledge about the future use of that tuple, i.e., about which other agent will read that tuple. Time uncoupling is enforced because tuples have their own life span, independent of the life cycle of the processes that generate them, or of the processes that may read them in the future. These two types of uncoupling enable time-disjoint processes and processes that do not know each other to coordinate seamlessly.

In addition, tuple-based coordination models provide for notable flexibility, which is an important requirement for open and dynamic software systems. Lacking a schema or a predefined organization, a tuple space does not restrict the format of the tuples it stores or the types of data it can contain, thus making it suitable for unexpected types of interactions. In addition, the scalability of a tuple space system is provided by the complete anonymity of tuple operations. No one has to keep track of connected processes to a specific tuple space. Thus, it is possible to conceive systems based on a multiplicity of independent localized tuple spaces [112], to enforce locality in interactions (see Fig. 3.3).

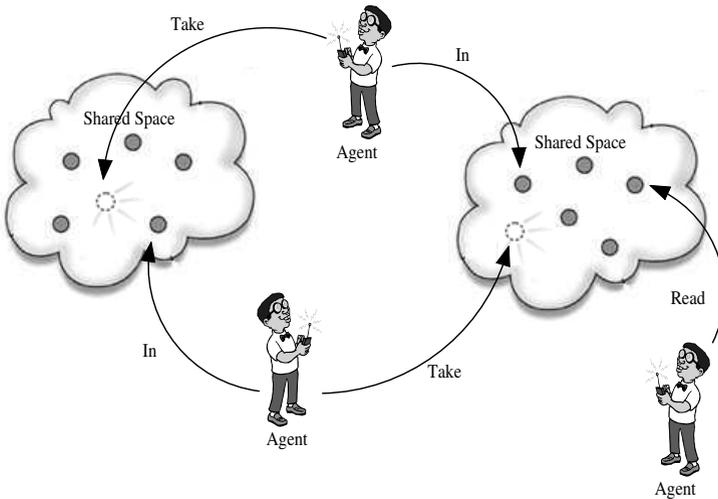


Fig. 3.3. Shared data space model: main operations on a tuple space, as provided in Javaspaces

Inadequacy

When adopting a tuple-based coordination model (i.e., when developing applications exploiting a middleware relying on tuple-based services), agents are no longer strictly coupled in their interactions, because shared tuple spaces mediate interactions promoting uncoupling. Also, a shared localized tuple space can be effectively used as a repository of local, contextual information. Still, such contextual information can only represent a strictly local description of the context that can hardly support the achievement of global coordination tasks.

In the case study, one can assume that the museum provides a set of tuple spaces, storing information such as a list of nearby art pieces as well as message tuples left by the other agents. Tourists can easily discover what art pieces are near them, but to locate a distant art piece they should query either a centralized shared tuple space or a multiplicity of localized tuple spaces, and agents have to internally synthesize all the information to compute the best route to the target. To meet with each other, tourists can build an internal representation of the other people's positioning by accessing several distributed data spaces, by reading tuples reporting about their presence, and then by locally computing a path in the museum. However, the availability of such information does not free them from the need for negotiating with each other to orchestrate movements.

The pseudo-code in Fig. 3.4 implements an agent performing the meeting application by accessing a shared (Javaspaces-like) tuple space middleware. Here, we have assumed the presence of a global space (whether provided by a specific server or obtained by merging agents' private spaces) on which all the agents can post and retrieve information in the form of tuples. It is rather easy to see that this kind of middleware is much more suited to manage open meeting groups than direct coordination middleware. In fact, the space uncouples the interaction between the agents in the meeting group (on rows 7-9, the agent retrieves tuples independently for who actually wrote them), and somehow provides a suitable means by which to access contextual information (i.e., the location of other tourists). However, in our opinion, a key problem is that agents are left alone in discovering relevant contextual information, in evaluating and possibly negotiating a meeting room, and in navigating across the museum (rows 12-14). This can lead to noticeable computational and communication burden.

It is fair to say that models like MARS [22] and TuCSon [121], by relying on *programmable* tuple spaces, are better suited for dealing with coordination. In fact, agents can program the middleware so that it can perform low-level coordination tasks on the agents' behalf. In the case study, for example, an agent could program the middleware to properly aggregate relevant information about other agent locations. In this way the agent would have access to the already aggregated information without the burden of doing it on its own.

```

01: // get the museum map
02: Tuple mapT = new Tuple("MUSEUM MAP")
03: museum = middleware.read(mapT)
04: // proceed with the meting
05: while not meet
06: // find where the other agents are
07: Tuple readT = new Tuple("MEETING", *, *)
08: Tuple[] locT = middleware.read(readT)
09: location[] = parse(locT[])
10: //compute the best room for the meting on the basis
11: // of the agents current location and museum map
12: room = computeBestRoom(museum, location[])
13: // move toward the meting room
14: goTo(museum, room)
15: // update my location
16: Tuple writeT = new Tuple("MEETING", this,this.getLocation())
17: middleware.write(writeT)
18: end while

```

Fig. 3.4. Pseudo-code of the meeting application with a shared data space middleware

3.3.3 Event-Based Models

Event-based models relying on *publish/subscribe* mechanisms make agents interact with each other by generating events and by reacting to events of interest, without having them to interact explicitly with each other. Typical infrastructures rooted on this model are Jedi [26] and Siena [23]. In [35] is presented a complete survey on this kind of model.

The model

A software event is a piece of data generated to indicate that something has occurred in a system, e.g., a user moved the mouse, or a datagram has arrived from the network, or a sensor has detected that someone is knocking at the door. All of these occurrences can be modeled as events, and information about what happened can be included as attributes in the events themselves.

Event-based programming, i.e., writing software systems in terms of event processing, is a commonly accepted practice: programming becomes a process of specifying “when this happens, do that.” This is particularly evident in graphics programming: if the mouse moved, move the cursor with it; if the user clicks this button, execute that procedure.

The simplicity of event-based programming is a key to its success: identify the events of interest; identify who (which processes/objects/agents) are

interested in handling those events; and identify what procedures event handlers have to execute upon the occurrence of specific events. Events and event handlers are coupled by exploiting a publish-subscribe schema: agents interact by publishing events and by subscribing to the classes of events they are interested in (subscriptions are associated with event handlers). An operating system component, called event dispatcher, is in charge of collecting subscriptions and events, and of triggering the proper reactions in event handlers (see Fig. 3.5).

In distributed systems, event-based programming can be supported by means of middleware services acting as event dispatchers, in charge of collecting subscriptions from agents interested in specific classes of events, and in charge of distributing events (i.e., in triggering reactions) to subscribers whenever appropriate. A variety of schemas can be conceived for subscriptions [26] (e.g., subscribing to specific classes of events, or to events whose attributes match a specific template, or to events occurring at specific sites and/or at specific times). A variety of solutions can also be conceived for how to implement event dispatchers (e.g., centralized vs. distributed) and for how to distribute events to subscribers (e.g., broadcasting vs. direct forwarding).

Whatever the solution adopted, event-based coordination models clearly provide for full uncoupling among interacting entities (the same as tuple space models), and also provide for an effective way to achieve contextual information at runtime (indeed, an event represents something that has happened in a context).

Inadequacy

The fact that event-based models promote both uncoupling (all interactions occur by asynchronous and typically anonymous events) and context-awareness (agents can be considered as embedded in an active environment capable of notifying them about what is happening) represent important features for large-scale, open, and dynamic systems.

In the case study example, a possible use of an event-based approach would be to have each tourist notify his movements across the building to the rest of the group. Notified agents can then easily obtain an updated picture of the current group distribution in a simpler and less expensive way than required by adopting shared data spaces. However, such information still relies on agents for the negotiating of coordinated movements and does not alleviate their computational tasks (i.e., in the case study, tourists still have to explicitly negotiate their movements, as from the pseudo-code in Fig. 3.6).

It is rather easy to see that here agents are indeed provided with an active middleware that notifies them about other agents' movements (the `react` method in row 9 is invoked by the middleware upon the detection of an agent movement). However, agents need to process these events on their own and, in this case, the `computeBestRoom` and `goTo` methods can be source of complexity, brittleness and inflexibility.

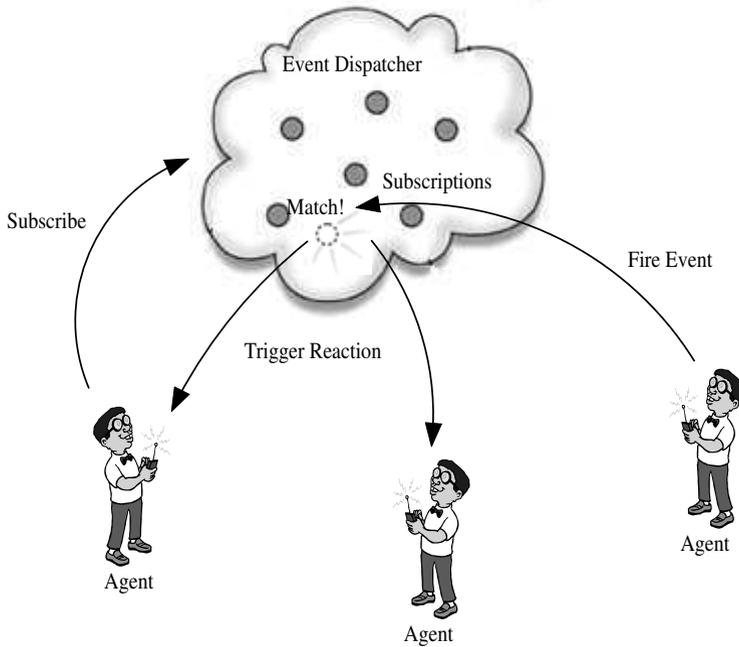


Fig. 3.5. Event based model: publish-subscribe operations

3.4 Requirements for Next-Generation Coordination Models and Systems

In this chapter, we have outlined the fundamental role of coordination for the engineering of adaptive self-organizing applications. At the same time, we have shown how current coordination models and infrastructures appear inadequate to the needs of emerging computing scenarios.

To summarize, the key characteristics that a proper coordination model should exhibit include

- the uncoupling of application agents, to properly facilitate spontaneous interactions and coordination activities in an open world;
- the integration of expressive means to acquire context-awareness, in order to facilitate agents in actually exploiting such information for their application purposes;
- the promotion of locality in interactions, to support scalability in large-scale and decentralized systems.

In the following chapter, we introduce field-based coordination as a potential candidate meeting the above requirements.

```
01: main() {
02:   // get the museum map
03:   museum = middleware.read(map)
04:   // subscribe to other agents movements
05:   Event newLocation = new Event("MEETING",*,*)
06:   middleware.subscribe(newLocation)
07: }
08:
09: react(Event newLocation) {
10:   // update my internal representation of the agents distribution
11:   location[].add(newLocation.source, newLocation.location)
12:   //compute the best room for the meting on the basis
13:   // of the agents current distribution and museum map
14:   room = computeBestRoom(museum, location[])
15:   // move toward the meting room
16:   goTo(museum, room)
17:   // notify other agents about my movement
18:   Event move = new Event("MEETING", this,this.getLocation())
19:   middleware.fireEvent(move)
20: }
```

Fig. 3.6. Pseudo-code of the meeting application with an event-based middleware



<http://www.springer.com/978-3-540-27968-6>

Field-Based Coordination for Pervasive Multiagent
Systems

Mamei, M.; Zambonelli, F.

2006, XII, 242 p. 127 illus., Hardcover

ISBN: 978-3-540-27968-6