ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria dei Sistemi e delle Tecnologie dell'Informazione

ADVANCED ROLE-BASED MODELS FOR ORGANISATION IN MAS COORDINATION INFRASTRUCTURES

Elaborato in:
SISTEMI INTELLIGENTI DISTRIBUITI LS

Tesi di Laurea di:
STEFANO RIFFELLI

Relatore:
Prof. ANDREA OMICINI

Correlatori:
Ing. ALESSANDRO RICCI
Ing. MIRKO VIROLI
Prof. ALAN WOOD

ANNO ACCADEMICO 2005–2006
SESSIONE II

to Angela

Contents

Introduction

In the engineering of complex software systems the agent paradigm is considered to be the most suitable level of abstraction, for planning, development and deployment. In general, in this characterization, an agent can be seen as an entity that autonomously develops one or more assignments (defined by its designer or planner), interacting in the environment in which it exists, through actions and perceptions. A complex system is modelled as a collection of agents (known as MAS, Multi-Agent System). MASs interact with the purpose to realize the goals of the system in its complex. Overall, a characteristic aspect of the MASs is the notion of *infrastructure*. Infrastructure is a software layer that furnishes different runtime services for the MASs. Another fundamental aspect of the MASs is the *organization*, which models the structure of the system and the structural relationships among its parts. Many important models in the MASs are based on the notion of role and groups (or society): in general a MAS is modelled as an organization divided into one or more groups, in which agents perform one or more roles. Typically, the roles determine what actions and perceptions are possible for the agents. Therefore, this approach allows capturing aspects related also to the safety, in particular to the control of access to the resources as well as other possible actions or interactions that the agents can perform.

This thesis aims to show and discuss the works on coordination languages and Role-Based models developed at the University of Bologna (Italy) and the University of York (UK).Initially, this thesis presents the overview of MAS (Multi-Agent System) and coordination languages. Subsequently, specific coordination languages developed at the University of Bologna (such as TuCSoN) and at the University of York (such as PyLinda and Lindacap) are discussed. Finally, Role-Based models in these coordination languages are examined.

More precisely, subsequent chapters are organised as follows:

1. **Agents and MAS overview**: the introduction of agent; definitions, typologies and environment of agents, description of MAS;

2. **The Concepts of 'Coordination Language'**: Coordination, Space-Based programming, the significances and categories of coordination languages;

3. **Tuple-Spaces Languages Design**: Linda model and some important Tuple-Spaces languages designs such as BeLinda, WCL, Java Spaces, TSpaces and others.

4. **Tuple-Space Languages from York & Cesena**: Tuple-Spaces languages from the University of York such as PyLinda (wich includes several of the more recently proposed extensions to Linda)and LindaCap (for coordination with multicapabilities) and from the University of Bologna (Cesena) such as TuCSoN (with programmable tuple centre behaviour and others important properties);

5. **RBAC model in MAS**: RBAC (Role-Based Access Control) model, comparing RBAC and ACL (Access Control Lists), RBAC in MAS, RBAC in TuCSoN, CNP as case study in Linda and in TuCSoN and finally a possible way to bring RBAC model in Lindacap.

Conclusion: a general evaluation and possible future works

Chapter 1

Agents and MAS overview

In the beginning of the new millennium, agents are among the most prominent and attractive technologies in computer science. The technologies, methods and theories of agents and multiagent systems are used in such diverse domains as information retrieval, user interfaces, electronic commerce, robotics, computer mediated collaboration, computer games, education and training as well as ubiquitous computing and social simulation. Not only are they a very promising technology, but also symbolise a new way of thinking, a conceptual paradigm for analyzing problems, designing systems and dealing with complexity, distribution, and interactivity. Furthermore, they provide a new perspective on computing and intelligence.

This chapter introduces the concept of agent as a fundamental abstraction to plan and create complex software system. The engineering of distributed systems calls for abstractions that encapsulate the control flow. Object-oriented paradigm encapsulates state and behaviour, but not a control flow. Moreover, control remains outside objects that are owned by a human designer. Method invocation does not allow to model concurrent activities as well as interaction and coordination of concurrent activities. Adopting a task-oriented approach over an object-oriented approach is advisable as the former allows identifying system basic tasks. Put differently, a task can be assigned to a single agent or a society of agents. In this way, two new abstractions are introduced: agent abstraction and coordination artifacts abstraction. *Agent abstraction* encapsulates the control flow responsible for carrying out an individual task whereas *Coordination artifacts* allow the shared means or tools used by the agent to fulfil a collective task. Agents and coordination artifacts need infrastructures that would both provide basic services to support agents and artifacts at runtime and

that would keep the abstraction alive [33]. While these infrastructures will be discussed in more detail in the second chapter, various definitions and typologies of agents as well as their environment will be examined in the first chapter, in which MAS and coordination will be also described.

1.1 Agent definition

A software agent is an abstraction, a logical model that describes software that acts for a user or other program in a relationship of agency. The idea is that agents are not strictly invoked for a task, but activate themselves. Simply put, an agent (software, hardware or otherwise) must exhibit behavioural qualities. In essence, this means perceiving the software environment through sensors and acting on the environment through actuators. This must be accomplished autonomously. Agents possess the ability to learn and to reason, logically or otherwise. They may discover facts and rules about the world and about others. They can also assess and argue these truths as well as alter their ontology.

Related and derived concepts include:

- *Intelligent agents*, in particular exhibiting some aspect of Artificial Intelligence, such as learning and reasoning
- *Autonomous agents*, capable of modifying the way in which they achieve their objectives
- *Distributed agents*, being executed on physically distinct machines
- *Multi-agent systems*, distributed agents that do not have the capabilities to achieve an objective alone and thus must communicate
- *Mobile agents*, agents that can relocate their execution onto different processors.

The term *'agent'* describes a software abstraction, an idea, or a concept, similar to OOP terms such as methods, functions, and objects. The concept of an agent provides a convenient and powerful way to describe a complex software entity that is capable of acting with a certain degree of autonomy in order to accomplish tasks on behalf of its user. But unlike objects, which are defined in terms of methods and attributes, an agent is defined in terms of its behavior. It is possible to classify the 'agent' notion into two types: weak and strong

The *Weak agent notion* denotes a hardware or software system that includes the following concepts:

- *autonomy*: agents have the capabilities of task selection, prioritization, goal-directed behaviour, decision-making without human intervention.
- *social ability*: agents are able to engage other components through some sort of communication and coordination, they may collaborate on a task.
- *reactivity*: agents perceive the context in which they operate and react to it appropriately.
- *pro-activeness*: the agent does not simply act in answer to its environment, but it is able to show a goal-directed behavior taking the initiative.

The *Strong agent notion* is especially used in Artificial Intelligence contexts, where an agent can be seen as a computerized system including the above concepts and not only. More precisely, an agent is implemented using other concepts usually applied to the human ones. For istance, an agent can be characterized using 'mental' notions such as knowledge, beliefs, intentions and obligations.

Thus, it is often considered as a particular software entity to the extent to which it acts in autonomous way: withdrawing information from the environment in which it is found, acting according to its knowledge base, exchanging information with other agents and with humans and, finally, taking the initiative to satisfy its own goals. The concept of Agent is more useful as a tool for system analysis than as a prescription.

The above mentioned concepts often relate well to the way one naturally think about complex tasks. Agents can be useful to model such tasks as they can be distinguished from objects for the followings principal motives:

- agents are more autonomous than objects;
- agents have flexible behaviour: reactive, proactive, social;
- agents have at least one thread of control but may have more.

Finally an agent can be implemented by a classic language (C, JAVA, PROLOG, ...) or by means of special agent language (Agent0, AOP, ...). By 'classic language', a 'computation language' is meant. In the second

chapter, the meaning of computation language will be uncovered and the difference between a computation language and a coordination language will be explored.

1.2 Agent typologies

In this section the above typologies of agents are shown in more detail.

Intelligent agents - The design of intelligent agents (or intelligent software agents) is a branch of artificial intelligence research. Capabilities of intelligent agents include:

- *ability to adapt*: adaptation implies sensing the environment and reconfiguring its response through the choice of alternative problem-solving-rules or algorithms or through the discovery of problem solving strategies. Adaptation may also include other aspects of the internal construction of agents, including recruiting processor or storage resources.

- *ability to learn*: learning may proceed through trial-and-error, implying a capability for introspection and the analysis of behaviour and success. Alternatively, learning may proceed by example and generalization, suggesting an ability to abstract and generalize.

Autonomous agents - Some software agents are thought to be autonomous as they are self-contained and capable of making independent decisions and taking actions aimed to satisfy internal goals based upon their perceived environment. This, however, is controversial. All software agents in important applications are closely supervised by people who start them up, monitor and continually modify their behaviour, and who shut them down when necessary.

Distributed agents - Since agents are well suited to include required resources in their description, they can be designed as very loosely coupled and it becomes easy to have them executed as independent threads and on distributed processors. They then become distributed agents and the considerations of distributed computing apply. Agent code is particularly easy to implement in a distributed fashion and should scale well.

In reality, all these agents can also be classified on the basis of other dimensions, including autonomy, cooperation and the ability to learn possible interactions with the environment. These dimensions are depicted in Fig.1.1 that shows different agent typologies [27].

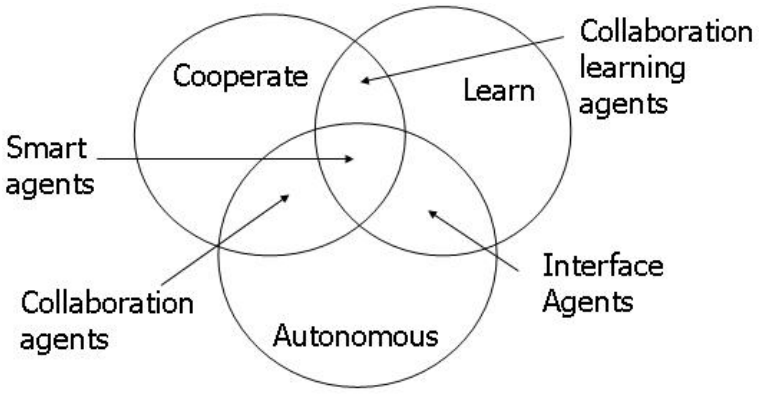

Figure 1.1: Agent typologies

Once it is decided what dimensions to use in preparing a classification, is possible to see different kinds of agents:

- Collaborative Agents
- Interface Agents
- Mobile Agents
- Information Agents
- Reactive Agents
- Hybrid Agents
- Smart Agents
- Heterogeneous Agents

Collaborative agents: are autonomous as they possess the ability to cooperate with other agents. Having autonomy, social ability and proactiveness, the agents are capable of learning these characteristics. Furthermore, they are able to act in an autonomous and rational way in open multi-agent systems and with temporal ties.

Interface agents: these agents are responsible for managing a dialogue with the consumer. They are autonomous and capable of learning, but generally they do not communicate with other agents, therefore often they are called personal assistants.

Mobile agents: agent code that moves itself, including its execution state, on to another processor, to continue execution there. This is also referred to as mobile code.

Information agents: These are software entities (intelligent agents) that may access one or multiple, distributed, and heterogeneous information sources available. They pro-actively acquire, mediate and maintain relevant information on behalf of their user(s) or other agents preferably just-in-time. In other words, information agents are supposed to cope with the difficulties associated with the information overload of the user. This implies their ability to semantically broker information by:

- providing pro-active resource discovery;
- resolving the information impedance of information consumers and providers;
- offering value-added information services and products to the user or other agents.

Reactive agents: the main characteristics of a reactive agent are:

- Reactive agents do not have internal symbolic models.
- Act by stimulus-response to the current state of the environment.
- Each reactive agent is simple and interacts with others in a basic way.
- Complex patterns of behavior emerge from their interaction.
- Benefits: robustness and fast response time
- Challenges: scalability

Hybrid agents: hybrid agents use different approaches to get the best properties of the each of above mentioned agents. Specifically, they aim for a quick response time of reactive agents for well known situations, not to mention their ability to generate new plans for unforeseen situations. For this purpose deliberative and reactive philosophies are often combined.

Smart agents: these agents are endowed with autonomy and the ability to collaborate and learn. Frequently, they are called intelligent agents. The termi 'intellagent' depends on the context in which it operates, pursuing its own objectives and performing its own assignments in order to optimize some measure of performance.

Heterogeneous Agent Systems: these are systems that integrate two or more agents belonging to two or more categories from the ones listed above. These systems allow for interoperability among these forms of software that operate in different domains through a suitable language of communication. When several agents interact they may form a MAS (multi-agent system). A system is only considered multiple agent if the agent under consideration acts cooperatively or competitively with another agent to achieve a task or a goal. Otherwise, this other agent is simply viewed as a stochastically behaving part of the system. Such agents will not possess all data or all methods available to achieve an objective (this can be referred to as 'limited viewpoint'), thus they are forced to collaborate with other agents. As with distributed agents, data is decentralized and execution is asynchronous. Finally, in Multi-Agent Systems, related fields include Distributed Artificial Intelligence (DAI) and distributed problem solving (DPS).

1.3 Agent environment

Agents are individual entities with social abilities. To an extent, they represent the world around them. Their ability to perceive and to change the surrounding world is limited. Establishing relationships with other agents is typical of them. Therefore, it can be said that agents live in society. The behaviour of an agent is often incomprehensible out of its social structure. When the agents are planned one cannot separate them from the environment in which they live: the idea of modelling the software system without modelling the environment seems to be conceptually wrong. Overall, agents and society live in different types of environment and these can be defined differently. Thus, it is useful to see the following definitions as referring to the way the environment appears from the point of view of the agent itself. More precisely, there are different kinds of environments [38]:

- *Observable or partially observable*: In order for an entity to be considered an agent some part of the environment must be observable. In some cases (particularly in software) the whole environment will be observable by the agent. While useful to the agent, this will generally be true only for relatively simple environments.

- *Deterministic or Stochastic*: An environment that is fully deterministic is one in which the subsequent state of the environment is wholly dependent on the preceding state and the actions of the agent. If an

element of interference or uncertainty occurs, then the environment is stochastic. A deterministic, though partially observable environment will appear to be stochastic to the agent.

- *Episodic or Sequential*: This refers to the agents' tasks in their environment. If none of the tasks performed by the agent rely upon past performance, and do not affect future performance, the environment is episodic. Otherwise it is sequential.

- *Static or Dynamic*: A static environment, as the name suggests, is one that does not change from one state to another when the agent is considering its course of action. In other words, the only changes to the environment are those caused by the agent itself. A dynamic environment changes: if an agent does not respond in a timely manner, it choses to perform no action.

- *Finite or Infinite*: This distinction refers to whether or not the environment is composed of a finite or infinite number of possible states. In practice, infinite environment is impossible to realize. However, if this number is extremely high, it becomes virtually infinite from the agents perspective.

The properties of the environment influence the way in which agents represent the world in which they exist and the way in which they plan their actions. The methodologies directed towards the agents must be capable of modelling the agents' environment from the first phases of the engineering process onwards. Further, they have to express dependencies among the agents and the same environment. Although the characteristics of an environment of agents are often not fully predetermined, they may be partially defined according to the system requirements. The environment of a multi-agent system needs to be subject to an engineering process, with the purpose of describing and shaping that environment.

1.4 MAS description

Many systems require planning, scheduling, coordination, communication, transport, simulation, and module integration technologies and often the research interest pertains to only a portion of the technology or to aggregate system performance. As the number of services available on the Internet increases, finding and using the best one for the job is harder and more tedious. Automation through artificial intelligence is often put forward as

the answer to this problem. Traditional approaches try to design intelligent behaviour from the top-down, the result being highly complicated systems with huge memory and computational requirements. By contrast, agent systems engineering is a bottom-up approach combining elements of distributed computing and artificial intelligence. Multi-agent systems promise to allievate huge memory problem through systems that communicate by means of high-level semantically ground messages [24]. The vision is that automated reasoning systems such as theorem provers can be employed to build intelligent applications that are able to solve problems in a goal directed and autonomous manner. A multi-agent system (MAS) is a system composed of several agents, collectively capable of reaching goals that are difficult to achieve by an individual agent or monolithic system. The study of multiagent systems (MAS) focuses on systems in which many intelligent agents interact with each other. The agents are considered to be autonomous entities, such as software programs or robots. Their interactions can be either cooperative or selfish. That is, the agents can share a common goal (e.g. an ant colony), or they can pursue their own interests (as in the free market economy). The exact nature of the agents is a matter of some controversy. They are sometimes claimed to be autonomous. For example a household floor cleaning robot can be autonomous in that it is dependent only on a human operator to start it up. On the other hand, in practice, all agents are under active human supervision. Furthermore, the more important the activities of the agent are to humans, the more supervision that they receive. In fact, autonomy is seldom desired. Instead interdependent systems are needed.

Some important researches performed at the University of Bologna (Information and Communication Technology Engineering) and at the University of York (Computer Science Departments) develop communications languages, interaction protocols, and agent architectures that facilitate the development of multiagent systems. For instance, some basic questions are how to program each ant in a colony in order to get them all to bring food to the nest in the most efficient manner, or how to set up rules so that a group of selfish agents will work together to accomplish a given task. So, multi-agent systems can manifest *self-organization* and complex behaviors even when the individual strategies of all their agents are simple. MAS can be claimed to include human agents as well. Human organizations and society in general can be considered an example of a multi-agent system. To share knowledge, agents can use Knowledge Query Manipulation Language (KQML) or FIPA's Agent Communication Language (ACL). An MAS has

the following *advantages* over a single agent or centralized approach:

- An MAS distributes computational resources and capabilities across a network of interconnected agents. Whereas a centralized system may be plagued by resource limitations, performance bottlenecks, or critical failures, an MAS is decentralized and thus does not suffer from the 'single point of failure' problem associated with centralized systems.
- An MAS allows for the interconnection and interoperation of multiple existing legacy systems. By building an agent wrapper around such systems, they can be incorporated into an agent society.
- An MAS models problems in terms of autonomous interacting component-agents, which is proving to be a more natural way of representing task allocation, team planning, user preferences, open environments, and so on.
- An MAS efficiently retrieves, filters, and globally coordinates information from sources that are spatially distributed.
- An MAS provides solutions in situations where expertise is spatially and temporally distributed.
- An MAS enhances overall system performance, specifically along the dimensions of computational efficiency, reliability, extensibility, robustness, maintainability, responsiveness, flexibility, and reuse.

1.5 Importance of policies

Of increasing importance is mobile-agent systems where agents move from one host to another in order to take advantage of services offered at each locality. Security is clearly a major issue here and each host may have specific rules and regulations on acceptable agent behaviour. This is an important problem in distributed computing where the concept of policies has been put to use to specify and enforce these rules [24]. *Policies* are essential for the successful management of any large organisation. As multi-agent technology advances, more and more agents will begin to populate agent systems. By their nature, agents exhibit autonomous, goal-directed problem solving behaviour, and this naturally leads to viewing a collection of agents as an agent society. In the same was as a human society needs

rules and regulations, so does an agent society. From another point of view, any large distributed system will need to be administrated in order to manage change. *Policies* are useful to specify not only how components in such a system relate to each other, but also which rules or laws must be observed by the components. Clearly agent systems can benefit from this management perspective in exactly the same way as traditional distributed systems. The multi-agent system design problem is often broken down into a set of entities and roles. Usually these roles are intended to cooperate in order to optimize some performance measure. Of course cooperation cannot always be guaranteed, since the system may be required to interface with external systems which have different agendas. Roles may be characterised by a set of responsibilities, which indicate not only what functions the role must perform, but also in which activities the role must not participate. Since policy languages are designed to express authorisations and/or event-condition-access rules, they are ideal for specifying role-responsibilities that are expected to change in the future.

In this chapter, definitions, typologies and environments of agents have been discussed. In the final part MAS description and the importance of policies have been shown in order to introduce 'coordination concepts'. Thus, in the next chapter, general concepts of 'coordination languages' are explained starting from coordination definitions and ending with the significance of a coordination language and its categories.

Chapter 2

General Concepts of 'Coordination Language'

The previous chapter gave a general understanding of the importance of coordination in complex systems such as MASs. This chapter briefly shows the importance of coordination and the significance of Space-Based programming. Finally, different meanings of the term 'coordination language' and categories of coordination languages are discussed.

2.1 Coordination

When there are interdependencies among agent activities, coordination of agent activities becomes necessary [25]. For instance, a simple situation of subproblem interdependencies occurs when agents are intending to work on the same or overlapping subproblems and have alternative methods or data that can be used to generate a solution. In these cases, *coordination* involves making decisions such as

- Should both agents work on the subproblem concurrently?
- If not, in what order?
- If only one agent is to solve the subproblem, which one?

Moreover, there are other cases in which *coordination* involves making decisions such as:

- When should each agent begin to solve its subproblem?

- When are intermediate results of a subproblem solution worthwhile transmitting?

When it is not possible to decompose the problem into a set of subproblems, additional interdependencies among subproblems arise and when there are these interdependencies, overall system performance is significantly affected by the different choices like the following:

- methods used to solve a subproblem,
- the order of solving the subproblems,
- the time at which a subproblem will be solved,
- what aspects of subproblem solutions are transmitted and to whom,
- etc...

The *usefulness of coordination* can be seen in the simple situation where one agent needs the results of a subproblem that another agent is solving. If it can be arranged that the producing agent will deliver in a timely fashion the desired result so that the consuming agent does not have to idle waiting for the results, then system performance is improved. The *coordination decision* then involves a complicated optimization problem of ordering tasks and selecting how to accomplish them that spans multiple agents. In general, the *importance of effective coordination* and the corresponding need for more sophisticated coordination mechanisms increases in some situations like the following:

- where there are complex subproblem interdependencies among agents,
- where there are time-pressures and resource bounds which preclude all goals of the system being solved in an optimal manner,
- where there are many choices available about how to solve a goal,
- where the goals being solved and the agents and resources available to solve them are changing over time,
- etc...

An interesting process intimately related to coordination is negotiation. *Negotiation* is the process of arriving at a state that is mutually agreeable to a set of agents and this process can be used as part of a multiagent

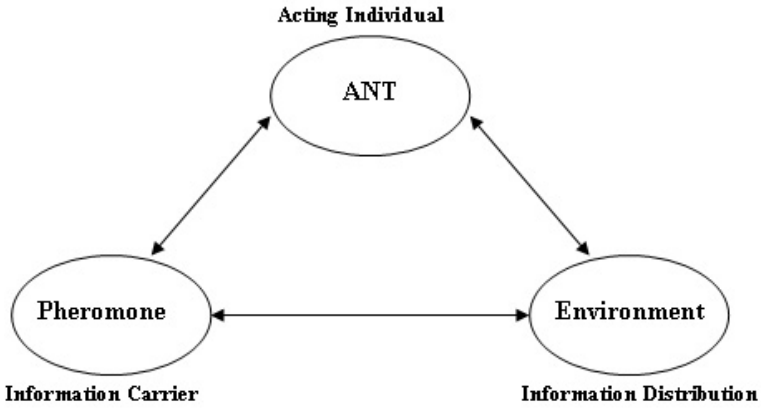

Figure 2.1: Elements of a Stigmergic System

coordination algorithm. Negotiation may also be used by agents to ensure that their solutions to subproblems are compatible. Effective coordination strategies will be very important for agents in next-generation multiagent systems. These agents will need to be highly adaptive due to their 'open' operating environments where the configuration and capabilities of other agents and network resources could change dynamically. There are different techniques used to create coordination between agents. How agents can achieve some form of coordination, depends on the chosen agent architecture. A specific architecture enables agents to interact or communicate in a certain way, or to take part in a society of agents with a certain social structure. Finally an agent should be able to choose to carry out an entire task itself, or it could elect to coordinate with appropriate agents to perform tasks collectively. So, systems consisting of a lot of such agents will need to be able to form and evolve higher order social structures such as teams and organizations to exploit collective efficiencies and to manage emerging situations.

A well-known example of coordination is stigmergic coordination. *Stigmergy* is an important method of communication in emergent systems in which the individual parts of the system communicate with one another by modifying their local environment. Stigmergy was introduced by studying biological insect societies and their concepts of interaction and information exchange. Investigations of biological insect societies show that these animals coordinate themselves by producing a dissipative field in their en-

vironment [19]. For instance, ants communicate with one another by laying down pheromones along their trails, so where ants go within and around their ant colony is a stigmergic system. Elements of a stigmergic system are depicted in Fig.1.2 that shows the insect as the acting individual, the pheromone as an information carrier - used to create a dissipation field - and the environment as a display and distribution mechanism for information. Stigmergy describes a form of asynchronous interaction and information exchange between agents mediated by an 'active' environment. Moreover, *Stigmergic coordination* has inspired and influenced algorithmic developments in application fields such as combinatorial optimization, routing in communication networks, exploratory data analysis, graph drawing and partitioning, manufacturing control etc.

2.2 Space-Based programming

In general, there are two main challenges facing any parallel programmer:

- How to divide the work among the available processors.
- Where to store the data and how to get it to processors that need it.

Two radically different approaches to these problems have emerged as the dominant parallel processing paradigms:

1. message passing
2. distributed data structures implemented in virtual shared memory.

Message passing focuses on the separate processes used to complete the overall computation. In this scheme, many concurrent processes are created, and all of the data involved in the calculation is distributed among them in some way. There is no shared data. When a process needs data held by another one, the second process must send it to the first one.

Distributed data structure programs are a second approach to parallel programming. This method decouples the data required for the calculation and the distinct simultaneously-executing processes which each perform a part of it, making them autonomous. Distributed data structure programs use a shared data space, with the individual processes reading data from it and placing results into it. The data structure is distributed in the sense that different parts of it can reside in different processors, but it looks like one single global memory space to the component processes [5].

In this second approach, '*Space-based programming*' heralds a new way of building distributed applications. Space architecture supplies a surprisingly compact model. Its inherent, minimalistic approach predisposes it to a wide range of applications while endowing it with the advantages of modularity, scalability, and source code economy. In the early 1980s, Gelernter (Yale) developed a programming language named Linda (see [17]) in order to support distributed applications. Linda was composed of a small set of operations in combination with a globally-accessible persistent store, the so-called 'tuple-space'. These operations form a coordination protocol, orthogonal to classical programming languages and thus readily incorporated within them. Research results have shown that a large number of problems in parallel and distributed computing can be neatly solved using this architecture.

By 'space', a virtual, distributed shared memory is meant. The necessary API essentially contains four methods: *write*, *read*, *take*, and *notify*. Hence, with 'object', any kind of entity is meant. Whereas *write* and *read* fetch and store objects in memory respectively, *take* removes its object following a read and *notify* announces changes to the space.

In a space implementation, the space itself synchronizes all of its participants. This is 'blackboard communication', in which the participants [2]:

- Know nothing about one another (their exchange is connectionless and anonymous).
- Need not share the same process or machine (interprocess).
- Are temporally independent of one another (asynchronous).

The space or space service provides the following services [16]:

- *Simultaneous transparent access*: The space transparently satisfies the simultaneous data access operations of multiple processes distributed throughout a network. Distributed data structures are easily constructed, with the details of their communication remaining hidden from application developers. The actual data delivery mechanism (replication, propagation) functions behind the scenes and varies from implementation to implementation.
- *Persistence*: The space harbors a storage mechanism that guarantees that objects remain accessible until a process explicitly removes them. It is also possible to have the space discard an object upon the expiry of a preset lease.

- *Associativity*: Objects are located in space using associative lookups. The read method thus asks for an object by specifying the contents of its data fields, perhaps using null values as wild cards. The space search engine then uses value and type comparisons to return the desired object.
- *Transactions*: A transaction model enforces the atomicity of one or more operations hosted by one or more spaces.
- *Code Transfer*: Objects are present in space in serialized form; i.e., as passive data. Once read, however, an object's data fields can be modified or its methods called.

The simplicity of the fundamental paradigm is crucial; it can very easily be grasped, rendering unnecessary complex interfaces with their correspondingly steep learning curves. The minimal API allows for greatly reduced development expenditures; by dissociating the 'senders' from the 'receivers', it renders the application logic connecting them simpler, more flexible, and more stable. This particularly supports the construction, analysis, testing, and re-usability of large applications.

The advantages of a space-based approach can thus be summarized:

- Faster development cycles.
- High scalability via the addition of cheap standard hardware.
- Shift of load from the web server to concurrent worker processes.
- Total decoupling of the interfaces.
- Distributed caching.
- Simple integration of heterogenous landscapes.

2.3 Significance of a coordination language

The concept of coordination is by no means limited to Computer Science. There are many definitions of what is coordination such as:

- Coordination is managing dependencies between activities.
- Coordination is the additional information processing performed when multiple, connected actors pursue goals that a single author pursuing the same goals would not perform.

- Coordination is the process of building programs by gluing together active pieces.
- etc.

Agent coordination needs to take into account in its reasoning the quantitative character of interactions among agent activities, the performance characteristics and resource demands of activities, the likely future activities of agents, resource loadings, hard and soft deadlines for completion of agent activities, and also more symbolic information such as the beliefs, desires and intentions of agents and the desirability - or undesirability - of certain joint states of agents being achieved. To satisfy agent coordination it needs a good coordination model. More precisely (see [18]), programming a distributed or parallel system can be seen as the combination of two distinct model:

- computation model
- coordination model

The *computation model* allows programmers to build a single computational activity: a single-threaded, step-at-a-time computation. The *coordination model* is the glue that binds separate activities into an ensemble. An ordinary computation language embodies some computation model. A coordination model can be viewed as a triple (E,L,M), where E represents the entities being coordinated, L the media used to coordinate the entities, and M the semantic framework [9]. Furthermore, a coordination language is the linguistic embodiment of a coordination model, offering facilities for controlling synchronisation, communication, creation and termination of computational activities. A coordination language embodies a coordination model; it provides operations to create computational activities and to support communication among them. In other words, one computation language plus one coordination language equals a complete programming system. Coordmation languages support a full range of ensembles, from parallel applications through distributed systems through time coordinated ensembles and a range of others. *Ensemble* is an important notion in this topic. More precisely, an ensemble is a collection of asynchronous activities that communicate. An *activity* is a program, process, thread or any agent capable in principle of simulating a Turing Machine. It could be a person or another whole ensemble again. An ensemble is a natural breeding ground for heterogeneity. A program that needs contributions from different machines, or different computing models will naturally be an ensemble.

Finally, ensembles are the best-adapted inhabitants of the evolving hardware environment and ensembles of all sorts dominate the near-term future of computing. Other important notions are orthogonality and generality [18]. By means *orthogonality* is possible and desirable to treat coordination as orthogonal to computation for purposes of building programs whereas by means *generality* is possible and desirable to define coordination in such a way that it applies to every asynchronous software ensemble. Generallty is suggested by orthogonality. The general-purpose computing language is a recognized, useful idea. It seems reasonable to posit a general-purpose coordination language as well. For istance, Java is a complete computation language, though it lacks intrinsic support for process creation and interprocess communication, dually Linda is a complete coordination language, although it offers no support for arbitrary computations. A computation language by itself is useless. A computation must communicate with its environment or it serves no purpose. A coordination language must allow one active agent to convey information to another whose state is evolving and unpredictable. The purpose of a coordination model and associated language is to provide a means of integrating a number of possibly heterogeneous components together, by interfacing with each component in such a way that the collective set forms a single application that can execute on and take advantage of parallel and distributed systems. The idea of a general-purpose coordination model directs attention to the fact that there is such a topic as ensemble building in general, and that it is a fundamental issue for computer science.

In general, a coordination metamodel is composed of [8, 33]:

- *Coordinables* as entities whose mutual interaction is ruled by the model (such as processes, threads, objects, users, agents, etc.)
- *Coordination Media* as abstractions enabling and ruling agent interactions or in other words, the core around which the components of the system are organised (such as tuple spaces, blackboards, etc.)
- *Coordination Laws* to define the behaviour of the coordination media in response to interaction events. Usually coordination laws are defined in terms of communication language (the syntax used to express and exchange data structures) and coordination language (set of interaction primitives and their semantics)

Finally, an important and typical model, the *Tuple Space model*, is depicted in Fig.2.1. In this model, the tuple space contains a bag of heterogeneous data objects/structures, and by means of a set of operations, is

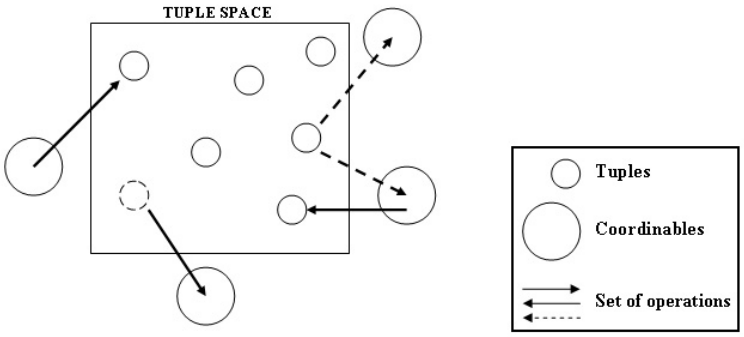

Figure 2.2: Tuple Space model

possible to put and retrieve tuples to/from the space. Tuples are the data structures of tuple space. A tuple is represented by a comma-separated list of items enclosed in parentheses, for instance: *('hello', 'day', 10, 'October', 2006)*. More precisely, tuple space, tuples and a set of operations play the role of coordination medium, communication language and coordination languages, respectively.

The tuplespace model provides a simple, yet powerful mechanism for interprocess communication and synchronization, which is the crux of parallel and distributed programming. A process with data to share 'generates' a tuple and places it into the Tuplespace. A process requiring data simply requests a tuple from the space, which is done by an associative match. Although not quite as efficient as message-passing systems, Tuplespace programs are typically easier to write and maintain, for the following reasons:

- *Destination uncoupling*: Most message-passing systems are partially anonymous: it is not necessary for the receiver of a message to identify the sender, but the sender must always specify the receiver. The creator of a tuple, however, requires no knowledge about the future use of that tuple, or its destination, so Tuplespace communication is fully anonymous.

- *Space uncoupling*: By using an associative addressing scheme for tuples rather than a physical one, Tuplespace is able to provide a globally shared data space to all processes, regardless of machine or platform boundaries.

- *Time uncoupling*: Tuples have their own life span, independent of the

processes that generated them, or any processes that may read them. This independence enables time-disjoint processes to communicate seamlessly.

Moreover, using a Tuplesapce model with the above generative communication properties [17], the same general purpose coordination language can be used in different coordination contexts, gluing different kinds of computations. Thus, others benefits [33] are implicated such as:

- *Heterogeneity*: Computation of heterogeneous computational models are glued all in the same coordination context.
- *Reusability*: It mean reusability (recycle-ability) in reusing application, implementation, tools and heterogeneous programmer expertise in the same coordination context.

Tuplespace extends message-passing systems with a simple data repository that features associative addressing. Conceptually it ranks above a pure message-passing system in terms of function but far below relational database systems, since most implementations do not include transactions, persistence, or any significant form of query facility [46]. Finally, a fundamental advantage of a Tuplespace system is flexibility.

2.4 Coordination language categories

There are a number of dimensions in which one can classify these models and languages, such as the kind of entities that are being coordinated, the underlying architectures assumed by the models, the semantics a model adheres to, issues of scalability, openess, etc. [32]. Mainly, there are two broad categories of coordination programming:

- data-driven
- control-driven

The main characteristic of the *data-driven* coordination models and languages is the fact that the state of the computation at any moment in time is defined in terms of both the values of the data being received or sent and the actual configuration of the coordinated components. In other words, a coordinator or coordinated process is responsible for both examining and manipulating data as well as for coordinating either itself and/or other processes by invoking the coordination mechanism each language provides.

It usually does mean that, at least stylistically and linguistically, there exists a mixture of coordination and computation code within a process definition. A data-driven coordination language typically offers some coordination primitives (coupled with a coordination metaphor) which are mixed within the purely computational part of the code. These primitives do encapsulate in a useful way the communication and configurational aspects of some computation, but must be used in conjunction with the purely computational manipulation of data associated with some process.

Instead, in the *control-driven* coordination models and languages there is an almost complete separation of coordination from computational concerns. Whereas in the case of the data-driven category, the coordination component is usually a set of primitives with predefined functionality which is used in connection with some 'host' computational language, in the control-driven category the coordination component is usually a fully-fledged language. Furthermore, whereas the data-driven category tends to be used mostly for parallelising computational problems and tends to coordinate data, the control-driven category tends to be used primarily for modelling systems. Some control-driven models examples are *IWIM* [3] and *Manifold* [4] (a stream-based communication model) , *Actors* [1] (a simple, asynchronous message passing model, with mailbox buffering) and *Ambient Calculus* [7] (with its mobile 'ambients' as collections of local running processes, which may contain other ambients, or 'capabilities').

Often, these two major families are distinguished due to two different and important notions:

- Most languages and models of the data-driven family have evolved around the notion of a *shared dataspace* which plays the dual role of being both a global data repository and an interprocess communication medium. Processes forming a computation either post and/or retrieve data from this medium. However, not all members of this family adhere to the Shared Dataspace concept; there are a few which use the message-passing metaphor or a limited form of Shared Dataspace in the form of common buffer areas or global synchronisation variables being manipulated concurrently by a number of processes.

- The control-driven family has evolved around some *process algebras notions* (such as CSP, CCS, etc.) of distinct entities communicating with the outside world by means of clearly defined interfaces, namely input/output ports, connected together in some appropriate fashion by means of streams or channels. In fact, the coordination paradigm offered by the control-driven family is sometimes characterised

as being channel-based as opposed to the medium-based notion of coordination supported by the data-driven family.

In this chapter, the main concepts of a coordination language have been discussed. The coordination languages developed by both the University of Bologna and the University of York mainly belong to the data-driven category. Thus, remembering that the final goal of this thesis is to examine Role-Based models in these coordination languages, the next chapter explores some languages such as Belinda, WCL, Java Spaces, Tspaces, etc. which all belong to the data-driven category. This is in order to present a global overview.

Chapter 3

Tuple-Space Language Design

As the previous chapter implies, it is possible to understand the importance of a coordination model. Thus, first of all, this chapter shows the Linda model. Subsequently, important coordination languages, including WCL, BeLinda, Java Spaces and Tspaces are discussed in order to reach global overview. More precisely, in each of these languages, the goals and the salient features are shown. In the end, other coordination languages are acknowledged.

It is important to point out that known coordination languages such as PyLinda and LindaCap (developed by University of York) and TuCSoN (developed by University of Bologna) are not discussed in this chapter. This is because they will be largely discussed in the next chapter.

3.1 Linda model

Linda has several characteristics which set it apart from other parallel programming environments:

- Linda augments the serial programming language (Java,C,Fortran and etc.). It does not replace it, nor make it obsolete. In this way, Linda builds on investments in existing programs.

- Linda parallel programs are portable, and they run on a large variety of parallel computer systems, including shared-memory computers, distributed memory computers, clusters, and networks of computers.

With few exceptions, Linda programs written for one environment run without change on another.

- Linda is easy to use. Conceptually, Linda implements parallelism via *tuple space* - the logically global memory (virtual shared memory) shown in section 2.2 - and a small number of simple but powerful operations on it. Tuple space and the operations that act on it are easy to understand and quickly mastered.

Although Linda can be used to implement both types of parallel programs - message passing and distributed data structures - shown in section 2.1, the most natural one for use with Linda is the distributed data structures using virtual shared memory. All interprocess communication is accomplished via this global data space. Processes never explicitly send messages to one another, but rather place data into the shared data space. When another process needs that data, it obtains it from the shared data space. Linda handles all data transfer operations, so the program need not worry about the exact mechanism by which it occurs. Linda operations require no additional overhead over any message passing scheme, and in fact sometimes are more efficient.

Linda or more precisely, the *Linda model* is a general model of parallel computing based on distributed data structures (although it can be used to implement message passing as well). Linda calls the shared data space *tuple space* (see section 2.2). Processes access tuple space via a small number of operations that Linda provides. For instance, if C-Linda is an implementation of the Linda model using the C programming language, parallel programs using C-Linda are written in C and incorporate these operations as necessary to access tuple space [5]. In this way:

- C-Linda functions as a coordination language, providing the tools and environment necessary to combine distinct processes into a complete parallel program.

- The parallel operations in C-Linda are orthogonal to C, providing complementary capabilities necessary to parallel programs.

- C-Linda programs make full use of standard C for computation and other non-parallel tasks.

- C-Linda enables these sequential operations to be divided among the available processors.

- Since C-Linda is implemented as a precompiler, C-Linda programs are essentially independent of the particular (native) C compiler used for final compilation and linking.

Linda programmers do not need to worry about how tuple space is set up, where it is physically located, or how data moves between it and running processes; all of this is managed by the Linda software system. Because of this, Linda is logically independent of system architecture, and Linda programs are portable across different architectures, whether they are shared memory computers, distributed memory computers, or networks of workstations. Data moves to and from tuple space as tuples.

Linda provides three operations for accessing tuple space

- Generation
  - *out*: Places a data tuple in tuple space
- Extraction
  - *in*: Removes a tuple from tuple space or blocking.
  - *rd*: Reads the values in a tuple in tuple space, leaving the tuple there, or blocking.

Both *rd* and *in* take a template as their argument. A template specifies what sort of tuple to retrieve. Like tuples, templates consist of a sequence of typed fields, some of which hold values and some of which hold placeholders for the data in the corresponding field of the matched tuple in tuple space. These placeholders begin with a question mark and are known as formals. If no matching tuple is present in tuple space, then both *rd* and *in* will wait until one appears; this is called *blocking*. The routine that called them will pause, waiting for them to return. When a matching tuple is found, variables used as formals in the template will be assigned the values in corresponding fields of the matched tuple.

A template matches a tuple when:

- They both have the same number of fields.
- The types, values, and length of all actuals (literal values) in the template are the same as the those of the corresponding fields in the tuple.
- The types and lengths of all formals in the template match the types and lengths of the corresponding fields in the tuple.

Generative communication results in a number of properties that distinguish Linda from other languages.

Summarizing (see [17]), those *properties* are:

- Distributed naming
- Time uncoupling
- Distributed sharing
- Continuation passing
- Structured names

Lindas weaknesses stem from its strengths: programmers are required to implement structures such as calling routines and queue-managing filters provided for them in other languages. Properties such as distributed naming and sharing make Linda inherently less secure than other languages: care must be taken to prevent unauthorized access to stored tuples.

In the next sections, some Tuplespace language based on Linda model such as BeLinda, WCL, Java Spaces and TSpaces will be shown.

3.2 BeLinda

BeLinda has been developed by Graduate Institute of Science and Technology of Beaverton (Oregon) and it is a benchmark for a portable software architecture based on Linda tuple space and it is portable, easy to implement and reflects performance measures for different primitives of parallel computation across several different machine types. An advantage of using a Linda based benchmark is that it is easily portable over a wide variety of hardware architectures [22]. The overall design and goals of BeLinda were influenced by the following *requirements* for a benchmark:

- be representative of actual applications.
- not artificially exclude a particular architecture or configuration.
- reduce to a single number to permit one-dimensional ranking.
- report enough details to be reproducible by an independent investigator.
- permit simple verification of correctness of results.

- use simple algorithms to fit into one page.

In order to meet these requirements:

1. BeLinda identifies a set of work primitives which are common across various parallel applications. The performance figures associated with these primitives can then be used as performance indicators for different parallel applications. More precisely, The work primitives, or constructs, identified within BeLinda fall into a number of *general categories*:

   - the basic Linda primitives: *in()*, *out()*, *rd()* and *eval()*. The efficiency of Linda on any architecture depends to a large extent on the implementation of these primitives.
   - primitives for communication and synchronization: *latency*, *contention*, multicast, *reverse multicast* and *message size*.
   - primitives for computation: the *add* and *multiply* times for integers, floats and double precision floats.
   - others issues: a number of these such as *evaluation of overlap of computation and communication*, *scaling-effect* and *blocked in()s* are also considered in BeLinda.

2. The use of Linda as a basis for BeLinda ensures that BeLinda shares the same high degree of portability as Linda.

3. Finally, BeLinda provides a statistical base of data, however, this data can be weighted and combined to produce a single performance figure.

3.3 WCL

In order to explain *WCL* coordination language, some Bonita concepts have to be introduced first. *Bonita* [A.Rowstron and A.Wood 1997] was developed in an attempt to overcome the problems of high latency in tuple space based coordination languages. The standard Linda primitives provide asynchronous inter-process communication, but provide synchronous communication with a tuple space. Bonita was developed to provide only asynchronous access to a tuple space. Therefore, an *in* is decomposed into a request for a tuple, and a receive tuple. The language provided primitives to check if a result was available and to block until a result tuple is

available. This means that not only it is possible to get the asynchronous tuple space access, but also mimic the traditional Linda primitives. This proved to be very powerful providing easy parallel tuple space access from a single process, and the ability to request multiple tuples and then act on them as they actually arrive. Also, it allows a form of *inp* primitive, which does not potentially hammer the run-time.

WCL was developed to take the concepts first explored in Bonita further. This attempted to integrate the concepts of tuple streaming and many different multiple tuple manipulating primitives. The tuple streaming included a sort of event model within the language, however, unfortunately this work was overtaken slightly by the work on JavaSpaces and TSpaces, which also included the concept. However, the WCL versions of the streaming was different, and more useful than the JavaSpaces approach. WCL supported a very wide range of primitives which supported many very different coordination patterns. The primitives emphasise efficiency and location transparency (of data, and agents) and concurrency transparency (ie. shared access to shared tuple spaces without interference). WCL does not specify any way to express computation and computation is provided by a host language into which WCL is embedded, and WCL and the run-time system currently supporting WCL is designed to be independent of the host languages used. WCL attempts to achieve language transparency. Bonita was the first and only other tuple space based coordination language to use asynchronous tuple space access, and provided a poor set of access primitive, which has been addressed in WCL. It is the first tuple space based coordination language to support the streaming of tuples. More precisely (see [35]), in WCL there are those kinds of primitives:

- *Basic tuple space access* primitives (both asynchronous and synchronous) wich are: *tscreate*(), *out_sync*(ts, tuple), *out_async*(ts, tuple), *in_sync*(ts, template), *rd_sync*(ts, template), *in_async*(ts, template), *rd_async*(ts, template), *check_async*(reply_id), *touch_sync*(ts, tuple), *touch_async*(ts, tuple), *cancel*(reply_id).

- *Streaming* and *Bulk* primitives which are: *move_sync*(ts_src, ts_dest, template), *move_async*(ts_src, ts_dest, template), *copy_sync*(ts_src, ts_dest, template), *copy_async*(ts_src, ts_dest, template) *bulk_in_async*(ts, template), *bulk_rd_async*(ts, template), *monitor*(ts, template)

The streaming of tuples is supported through the use of the *monitor*, *touch_sync*, *touch_async*, *bulk_in_async* and *bulk_rd_async* primitives.

More in detail:

- *monitor* places a monitor on a tuple space for tuples that match the template template. All the tuples within the tuple space ts that match the template template are streamed back to the user agent, and any tuples subsequently inserted are also sent to the user agent. A *monitor* is guaranteed to match and return any tuple inserted that matches the template, regardless of whether the tuple is also matched by another primitive.

- *touch_sync* inserts the tuple into the tuple space *ts* and then attempts to destructively read it from the tuple space. This primitive is synchronous and when the primitive returns the tuple tuple is no longer present in the tuple space. When the tuple is inserted in the tuple space if there are other primitives blocked that could match the tuple being inserted then they compete for the inserted tuple and the winner is non-deterministic.

- *touch_async* is the same as touch sync except the primitive does not block.

- *bulk_in_async* is used to retrieve all tuples that match the template template from the tuple space *ts*. The matched tuples are removed from the tuple space *ts*. The primitive returns a reply identifier *reply_id* which is then used in conjunction with the *check_async* and *check_sync* primitives to retrieve the matched tuples, one tuple at a time. The tuples can be considered as being streamed to the user agent. It should be noted that there is no way to determine the number of tuples returned, but this primitive can be used in conjunction with the *copy* and *move* family of primitives.

- *bulk_rd_async* is the same as *bulk_in_async* except the returned tuples are not removed from the tuple space it *ts*.

The bulk primitives of WCL: *move_sync*, *copy_sync*, *move_async* and *copy_async* are based on the synchronous primitives *collect* [Butcher et al. 1994] and *copy_collect* [A.Rowstron and A.Wood] proposed for inclusion in Linda. These are called the bulk primitives because they manipulate more than one tuple at a time. These primitives move multiple tuples from one tuple space to another tuple space.

To finish, it is important to point out that WCL is a lower level coordination language compared to Linda. This is due to the fact that the Linda primitives are not good for use in high latency networks, and that they do not support all the coordination constructs that are required for a

more open coordination style. WCL has been used in a number of applications. Also these, have demonstrated the flexibility of WCL in supporting applications where the number of participating agents change dynamically.

3.4 Java Spaces

JavaSpaces technology is a service specification from Sun Microsystems, facilitates building distributed applications for the Internet and Intranets. The JavaSpaces model involves persistent object exchange 'areas' in which remote processes can coordinate their actions and exchange data. It provides a ubiquitous, cross-platform framework for distributed computing, emerging as a key technology in this expanding field as well as it provides a distributed persistence and object exchange mechanism for Java objects. It can be used to store the system state and implement distributed algorithms. In a JavaSpace all communication partners (peers) communicate by sharing state. It is an implementation of the Tuple spaces idea. JavaSpaces is used when someone wants to achieve scalability and availability and at the same time reduce the complexity of the overall system. JavaSpaces technology was a precursor to the Java Jini technology, which has not been a commercial success, although it has found and kept new users over the years. However some vendors are now offering JavaSpaces-based products as a way to provide distributed reliable persistence for non-Jini environments. All too often, people focus on having for example, one database (or a JavaSpace, a message queue or whatever as well) containing all data and trying to make that single thing as fast as possible. It is often better to have more than one database and partition data appropriately between them. Therefore, when more than one database (or a JavaSpace, a message queue or whatever as well) is needed, some kind of factory or broker is needed to allow for easy management of multiple instances as well. JINI provides a natural set of patterns for this kind of partitioning/splitting. Creating a model that can cope with additional JavaSpaces is relatively straightforward. In cases where one is getting unprocessed 'entrys' building up in a JavaSpace, the JINI lookup model and the JavaSpaces API make it easy to add more machines to speed up emptying.

When reliability/availability are requirements, the standard way is to use clustering, but that is not always necessary. In many cases, it is possible to design simple hashing algorithms to scatter data across multiple JavaSpaces. Once this has been done, it is possible to then scatter to groups of JavaSpaces to achieve replication. For this approach to

be performant, it is often necessary to have entries be uniquely identifiable by means the existence of such primitives `asnet.jini.id.Uuid` and `net.jini.id.UuidFactory and SecureRandom`. An *entry* is an object that is stored in a space. To be an entry, an object just needs to implement the entry interface [16]. Once unique identifiers for entries are held, one can employ simple voting mechanisms to determine whether an operation should succeed or not. As an example, let is define a message entry that it is possible to write into a space:

```
import net.jini.core.entry.Entry;

public class Message implements Entry {
   public String content;

   // a no-arg constructor
   public Message() {
   }
}
```

In this example is defined a Message class with a string field that will hold the content of the message. To use this class with spaces, it needs to implement the interface `net.jini.core.entry.Entry`, which is found in the package `net.jini.core.entry`. It is important to point out that Entry is a *marker interface*; in other words, the interface contains no constants or methods and therefore requires no special work to implement, other than adding implements Entry to the class definition. Besides implementing the Entry interface, there are a few other conventions that our entries must follow. For instance:

- an entry must have a public constructor that takes no arguments,
- fields of an entry should be declared public
- fields of an entry must contain references to objects, rather than primitive types
- etc.

However, these requirements a part, an entry is like any other Java class. In other words, is possible to instantiate it, invoke its methods, and assign values to its public fields. Now, let is see what operations are

available for interacting with entries in spaces. To interact with a space, it need to obtain access to an object that implements the JavaSpace interface. There are many ways of obtaining access to such an object. The following methods are mainly the methods of the JavaSpaces API and are supplied by any object that implements the JavaSpace interface:

- *write*: places one copy of an entry into a space. If called multiple times with the same entry, then multiple copies of the entry are written into the space.
- *read*: takes an entry that is used as a template and returns a copy of an object in the space that matches the template. If no matching objects are in the space, then read may wait a user-specified amount of time until a matching entry arrives in the space.
- *take*: Works like read, except that the matching entry is removed from the space and returned as the result of the take.

In addition, the JavaSpace interface supplies the following methods, which can be useful in many applications:

- *notify*: Takes a template and an object and asks the space to notify the object whenever entries that match the template are added to the space. This notification mechanism, which is built on Jini's distributed event model, is useful when is required to interact with a space using a reactive style of programming.
- *snapshot*: Provides a method of minimizing the serialization that occurs whenever entries or templates are used; it is possible to use snapshot to optimize certain entry usage patterns in own applications as well.

3.5 TSpaces

TSpaces is network middleware for the new age of ubiquitous computing from IBM. A succinct description of TSpaces would be, a network communication buffer with database capabilities. It enables communication between applications and devices in a network of heterogeneous computers and operating systems. TSpaces provides group communication services, database services, URL-based file transfer services, and event notification services. It is implemented in the Java programming language and thus it automatically possesses network ubiquity through platform independence,

as well as a standard type representation for all datatypes.It extends the basic Linda Tuplespace framework with real data management and the ability to download both new datatypes and new semantic functionality [46].

The TSpaces system is appropriate for application that has distribution or data storage requirements. It can perform many of the duties of a relational database system without imposing an overly restrictive (and primitive) type system, a rigid schema, a clumsy user interface or a severe runtime memory requirement. In a sense, it is a database system for the common everyday computing device that does not generate complex SQL queries, but one that needs reliable storage that is network accessible.

In the TSpaces world, a common computing environment, with access to all possible network services, is surprisingly easy to build. A set of applications, written in Java, map the system-specific service or application to a standard tuple representation. Then, any client from any platform can generate a tuple and send it to a TSpaces server. Applications, listening for specific tuples, pick up jobs when they see them and execute them. This is all fairly fast, since it is implemented with an underlying memory-resident database system.

The salient *features of the TSpaces* system are:

- *Tuplespace operator superset*: TSpaces implements the standard set of Tuplespace operators: *read*, *in* (take), and *out* (write). In addition, it includes both blocking and nonblocking versions of take and read, set-oriented operators such as *scan* and *consumingscan*, and others operators such as *rhonda*.

- *Dynamically modifiable behavior*: In addition to the expanded set of built-in operators, TSpaces allows new operators to be defined dynamically. Applications can define new datatypes and new operators that are downloaded into the TSpaces server and used immediately. This is in contrast to relational database systems that have limited datatype support and limited dynamic function.

- *Persistent data repository*: TSpaces employs a real data management layer, with functions similar to heavyweight relational database systems, to manage its data. [31] TSpaces operations are performed in a transactional context that ensures the integrity of the data.

- *Database indexing and query capability*: The TSpaces data manager indexes all tagged data for highly efficient retrieval. The expanded query capability provides applications with the tools to probe the

data with detailed queries, while still maintaining a simple, easy-to-use interface.

- *Access controls*: Users can establish security policies by setting user and group permissions on a Tuplespace basis.
- *Event notification*: Applications can register to be notified of events as they happen in the TSpaces server.

TSpaces embodies the convenience of the Tuplespace communication model, the robust reliability and trustworthiness of a database system, and the platform independence of the Java programming language. Furthermore, it not only serves as the interface to other programs and distributed components, it is a fully functional data repository that can act as the primary data store or a caching front end to a corporate data warehouse.

In this chapter only some aspects of important coordination languages have been summarised, but obviously, there are many others coordination model/languages based on tuple space paradigm such as Bauhaus, Bonita, GigaSpaces, Klaim, Laura, Ligia, Limbo, LIME, LuaTS, MAGNET, MARS, Melinda, Paradise, Sonia, York Linda 1, York Linda 2 , etc.

In the next chapter, Tuple-Space languages implementations such as PyLinda & Lindacap (developed by University of York) and TuCSoN (developed by University of Bologna) are discussed.

Chapter 4

Tuple-Space Languages from York & Cesena

This chapter deepens PyLinda & Lindacap languages (developed by the University of York) and TuCSoN (developed by the University of York). More precisely some PyLinda properties and coordination with multicapabilities in Lindacap will be initially shown. Although PyLinda is a simple implementation of a Linda system, it also includes several of the more recently proposed extensions to Linda. Instead, Lindacap is a Linda-like capability-based system, the implementation of which involves some particular characteristics. Subsequenly, the TuCSoN model will be shown, its coordination space, its communication language and the use of TuCSoN from Java and from tuProlog. Moreover, some TuCSoN tools such as `CLIAgent` and `Inspector` are discussed. While the former allows users to invoke the commands of the TuCSoN coordination language, the latter allows them to observe and debug the communication state and behaviour of a tuple centre. Finally, bearing in mind the fact that tuple centre behaviour can be programmed by means of ReSpecT (defining the coordination laws ruling agent interaction), basic parts of this languages are shown.

4.1 PyLinda

PyLinda is a simple implementation of a Linda system, however it also includes several of the more recently proposed extensions to Linda [44]. Firstly, each computer in Linda network is required to have a server running on it. The server coordinates access to any tuplespaces it stores and controls any client processes running locally to it. A way to create a network of one

or more Linda servers, can be achieved by executng the following command on one node of the network:

```
linda_server.py
```

In order to add a new server to the Linda network run the server with the `-c` option followed by an ip address or a dns name of a computer already connected to the linda network. Thus, it is possible to proceed, executing:

```
linda_server.py -c<ip address or dns name>
```

on each of the other nodes where the ip address or dns name is for a node where the server has already been started. Then, there are some server options:

1. to change port
2. to reduce the chance of a security flaw

1) For instance, if the default port (2102) it is not suitable for some reason, it is possible to change the port - that the server listens on - in this way:

```
linda_server.py -s<new port>
linda_server.py --server-port=<new port>
```

Now, the server run on a different port. In order to connect to a server running on a different port, the following commands are required:

```
linda_server.py -cvenice -p<new port>
linda_server.py -cvenice --connect-port=<new port>
```

If the server is running on a different port then any client programs will need to connect on that port - this can be done by passing the new port to the connect function.

```
linda.connect(2103)
```

2) This other server option prevents any process or server that is not running - on the same computer - from connecting. The following option may be come useful to prevent any security problems from running

PyLinda; in this way PyLinda is not completely secure, but certainly the chance of a security flaw being found is reduced:

```
linda_server.py -l
linda_server.py --localbind
```

Now, any Linda client, first of all, have to connect to the Linda network by importing the Linda package and then calling the connect function as follow:

```
import linda
linda.connect()
```

The *Universal TupleSpace* is accesible by all Linda client. However, UTS is often used as a way for various process to contact each other; but through others tuplespaces, the majority of communication should be done. By creating an instance of the TupleSpace class, is possible to create new tuplespace:

```
ts = linda.TupleSpace()
```

In the follow, is shown an example of the master process that want to output the TupleSpace reference to the UTS where the slave processes can read it.

The master process:

```
linda.universe._out(("My identifying string", ts))
```

The slave process:

```
ts = linda.universe._rd(("My identifying string", linda.TupleSpace))
```

In order to perform some operations on this TupleSpace, all the available operations in PyLinda are listed in Tab.4.1.

PUBLIC MEMBER FUNCTIONS IN PyLinda	
TupleSpace ()	The TupleSpace constructor
~TupleSpace ()	The TupleSpace destructor
_out (Tuple t)	Output a tuple to the tuplespace
_in (Template t)	Destructively read a tuple from the tuplespace
_rd (Template t)	Non-destructively read a tuple from the tuplespace
_inp (Template t)	Destructively read a tuple from the tuplespace and non-blocking
_rdp (Template t)	Non-destructively read a tuple from the tuplespace and non-bloking
collect (TupleSpace dest, Template t)	Move all tuples matching the template to another tuplespace
copy_collect (TupleSpace dest, Template t)	Copy all tuples matching the template to another tuplespace

Table 4.1: Public memeber functions in PyLinda.

Although PyLinda is a simple implementation of a Linda system, it also includes some extensions to Linda in the form of:

- *Multiple tuple spaces*: The initial specification of Linda had a single, global tuplespace. One of the most obvious extensions to this model is to allow multiple tuplespace. In general, there are two models to obtain multiple tuple spaces: *Nested Tuplespaces* or *Flat Tuplespaces*. However the trend has been towards flat tuplespaces that means multiple disjoint tuplespace, probably because the flat model is more flexible and the added structure of the nested model is useful in some situation but counterproductive in others.

- *Garbage collection*: In an open implementation of Linda, with multiple tuplespaces, a major problem is memory usage. If a system implements the bulk tuple operations then a common design pattern is to create a new tuplespace, copy some tuples to it to count how

many there are and then lose the handle to the tuplespace. If no garbage collection is implemented then this tuplespace will remain in existence indefinitely. It it obvious that this situation will quickly cause the system to run out of memory and fail. Local garbage collection is a well understood problem, and has been implemented in many languages, such as Java and Python however expanding it to a distributed system is a challenge - In his PhD thesis [10] Menezes describes a system to perform garbage collection in the Linda-like system, Ligia.

- *Non-blocking primitives*: One of the features of the original Yale version of Linda was duplicates of the *in* and *rd* primitives, *inp* and *rdp*, that instead of blocking if no tuples match they returned a special value indicating no match. This has the significant problem of breaking the time independence of a Linda program. This results in a program being dependent on when it reaches each point in the program, because sometimes the call may return a tuple, other times it may not and there is no way to know what will happen. With the blocking versions the call will return only if there is a tuple that matches, otherwise it will never return.

- *Bulk tuple operations*: The classical Linda model suffers from a problem known as the *multiple rd problem*. If a process wishes to perform a non-destructive read on a subset of tuples then in the classic model all the tuples must be destructively read, processed and then returned to the tuplespace. The multiple read problem comes from the non-deterministic nature of Linda. The semantics of Linda do not specify which tuple will be received if there is more than one tuple that matches the template being matched. In [37] a pair of new primitives are proposed collect and copy-collect. Given two tuplespace $t1$ and $t2$ *collect(*$t1$*,* $t2$*, template)* moves all tuples that match template from $t1$ to $t2$. *copy-collect* is similar except that it copies rather than moves the tuples between the two tuplespace. Both primitives return a integer representing the number of tuples that were affected by the operation. A side benefit of using these new primitives is that the efficiency of the system is increased.

Finally, it is important to point out that rather than just having totally separate tuplespace other possibilities are the use of *scopes* [26] and of *multicapabilities* [43]. Intuitively a scope is an overlapping tuplespace that gives a view on some or all of a group of tuples. The ability to partition a

tuplespace gives the ability to easily replicate the functionality of the *collect* and *copy-collect* primitives. Using scopes to partition a tuplespace improves on the *collect* primitive by moving tuples between different scopes of the same tuplespace, this allows the process to solve the 'multiple rd' problem, and still allow other processes that have access to the whole tuplespace access to the tuples at all times. Multicapabilities provide a partitioning of a tuple-space and they will be discussed in the next section.

4.2 Lindacap

Lindacap is a Linda-like capability-based system that assumes an implementation having the following main characteristics [43]:

- There is a public universal tuple-space (UTS) accessible by all.
- All agent communications are done solely in terms of tuple-space operations.
- Every tuple-space, with the exception of the universal tuple-space, is explicitly created by agents in the system.
- tuple-spaces in the kernel are at structured. Every tuple-space is created at the same level, that is, tuple-spaces cannot be created inside others.
- Capabilities (handles/references) for tuple-spaces are first class objects.

One aspect of having a finer control is to be able to restrict what methods an agent is allowed to invoke on an object. A *capability* [20] is an unforgeable 'ticket' given to an agent that specifies which kind of actions on a certain object are permitted to the holder of the capability. In a more general way, capabilities are 'visibility' filters to create a more refined control over agents' actions on objects in the system. In the Linda context, tuple-spaces are uniquely identifiable -therefore, they can be referenced by capabilities- whereas tuples are anonymous: they can only be referred to using associative matching. A solution to this problem can be obtained by using multicapabilities. *Multicapabilities* extend capabilities for a collection of objects, rather than the traditional notion of a unicapability referring to a single named object. Whereas a unicapability allows its holder to operate in a certain way on the object it refers to, a permission in a multicapability allows the operation on any element within the group,

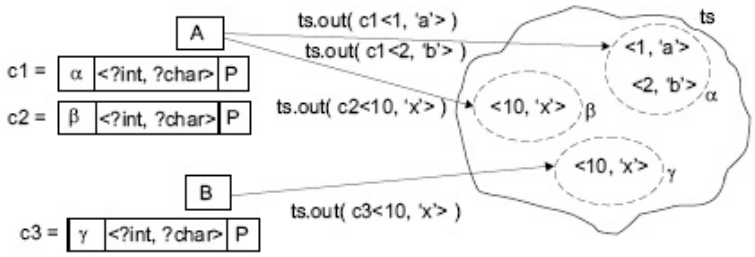

Figure 4.1: TS operations using multicapabilities [43]

not on the entire group referred to by the multicapability. In other words, tuples are classified by their multicapabilities.

For instance, Figure 3.1 shows an example of TS operations using multicapabilities. More precisely, whereas agent A makes two different requests for the multicapabilities *c1* and *c2*, agent B requests a multicapability *c3*. It is important to point out that when an agent wishes to perform any operation in the *ts*, before referring to a class of tuples, it must first request an appropriate multicapability from the kernel. Hence, the kernel represents the totality of the Linda 'middleware'. It is possible to notice that a multicapability is composed by three fields:

1. Unique multicapability name (such as α, β and γ)
2. Template (such as `<?int, ?char>`)
3. A set of permissions P granting the agent full rights to perform all operations on tuples which match the template. Using the multicapabilities they possess, the agents can perform operations (within those permitted by P)

Therefore, multicapabilities provide a partitioning of a tuple-space, thus enabling certain operations to be performed on tuples of a specific group, but not on those of another group, even though both groups have the same template.

The basic input/output primitives in Lindacap extends the basic primitives of Linda with multicapabilities as additional parameters (see [43]). The *primitives* are:

- `void` ***out*** `(TScap` ***ts***`, Multicap` ***mcap***`, Tuple` ***tup*** `)`: Given that the `TScap` and `Multicap` grants the ***out*** permission, this method

stores a Tuple in the region specified by `Multicap`, creating a new region if none exists of that multicapability.

- `Tuple rd (TScap ts, Multicap mcap, Template tmp )`: If both the `TScap` and `Multicap` grants the `rd` permission, this method searches for a Tuple matching the template `tmp` within the region specified by `Multicap` in the tuple-space `ts` . If a matching tuple is found, it is returned; otherwise, the method blocks until one becomes available.

- `Tuple in (TScap ts, Multicap mcap, Template tmp )`: This method executes in a similar way to `rd` , except that the matching tuple is removed from the `mcap` region and returned.

Lindacap also provides predicated primitives inp and rdp [21, 36], which are similar to the descriptions of in and rd above, except for their unblocking properties.

4.3 TuCSoN

TuCSoN (Tuple Centres Spread over Networks) is an infrastructure providing services enabling the communication and coordination of *agents* (as distributed / concurrent independent software components) [12]. TuCSoN infrastructure supports agent communication and coordination providing *tuple centres*, which are shared & reactive information spaces, distributed over the infrastructure nodes, where agents insert, retrieve, read information in the form of tuple, ordered collections of heterogeneous information chunks [11]. More specifically, tuple centres are programmable tuple spaces. In other words, tuple centres specialise and extend the basic tuple space keeping generative communication benefits. Moreover, while the behaviour of a tuple space in response to communication events is fixed, the behaviour of a tuple centre can be tailored to the application needs by defining a set of *specification tuples* expressed in the ReSpecT language, which define how a tuple centre should react to incoming/outgoing communication events [29]. So, such as tuple spaces, agents access tuple centres associatively by writing, reading, and consuming tuples via simple communication operations (*out, rd, in, inp, rdp*) but differently from tuple spaces, tuple centres can be programmed with reactions so as to encapsulate coordination laws directly in the coordination media. TuCSoN coordination space is composed by different TuCSoN node. Each node can englobe from *1* to *n* tuple centres. This structure is possible to see in Fig.1, in which TuCSoN

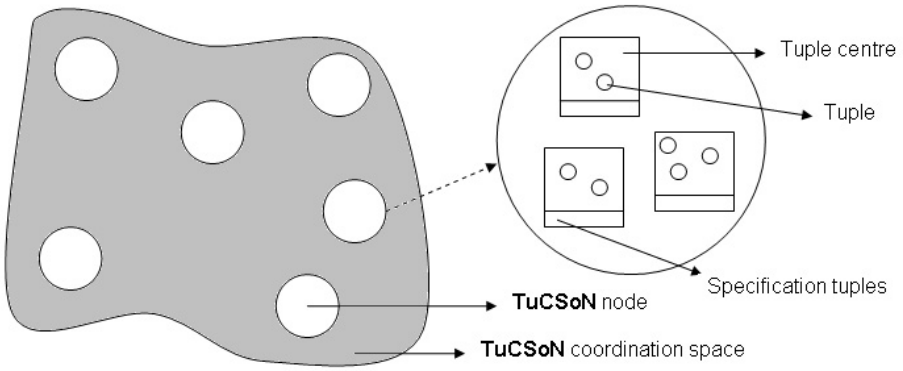

Figure 4.2: TuCSoN coordination model

coordination model is depicted. It is important to notice that each tuple centre possesses a new part - in comparison with tuple space - in order to specify a set of *specification tuples*. Moreover, a tuple centre has one own univocal name on the TuCSoN node in which it is.

To refer to a certain tuple centre, one will have to point out:

<Tuple Centre Name> @ <IP Address>

For instance:

```
newtuplecentre @ 127.0.0.1
```

TuCSoN also defines communication primitives that the agent has to englobe for using tuple centre. The agents interact exchanging tuples through the tuple centres using its primitive of communication shown in Tab.4.2 [11].

TuCSoN **OPERATIONS**	
out	to put the specified tuple in the specified tuple centre
in	to remove a tuple matching the specified tuple template from the specified tuple centre
rd	to read a tuple matching the specified tuple template from the specified tuple centre
inp	to remove, if present, a tuple matching the specified tuple template from the specified tuple centre (if not present the operation fails)
rdp	to read, if present, a tuple matching the specified tuple template from the specified tuple centre (if not present the operation fails)
set_spec	to set the behaviour of the specified tuple centre; tuple must be a valid ReSpecT program, in the form of a text (understood as a string atom) or list of reaction
get_spec	to get current behaviour of the specified tuple centre

Table 4.2: TuCSoN operations

Therefore, tuple centres are not only programmable behaviour but they have the following features [33] as well:

- *Adaptable at runtime*: Tuple centre behaviour can be changed/adapted dynamically, at runtime, by reprogramming the *artifact* (by means `set_spec`). Artefact is a Computational Entity without its own control flow and which can be used by agents for their own purposes.

- *Inspectable at runtime*: Tuple centre behaviour can be inspected dynamically, at runtime (by means `get_spec`)

The commands of the TuCSoN coordination language are in the following general form:

```
TCName @ TCAddress ?  op (Tuple )
```

where:

- `TCName` is the tuple centre name: any string may be used, including strings with spaces (in that case enclosed by apices).
- `TCAddress` is the tuple centre address: it must be a valid host name or IP address.
- `op` is the operation to be performed listed in Tab.4.2
- `Tuple` is the information content involved in the operation: it must be a valid logic tuple or tuple template (that is, roughly speaking, a Prolog term)

However, it is possible to write commands of the TuCSoN coordination language, in these others forms:

`TCName ? op (Tuple )` (In local resource)

`op (Tuple )` (In *default* tuple centre)

The quickest way to interact with the infrastructure is through the `CLIAgent` tool, which allows users to invoke the commands of the TuCSoN coordination language, providing a simple interface to invoke directly tuple centre coordination language primitives (actions) (`out,in,rd,inp,rdp set_spec` and `get_spec`).

Firstly, the infrastructure is set up by installing at least one node. Once TuCSoN in the *dir* directory is properly installed, type:

```
java -cp dir /tucson.jar alice.tucson.runtime.Node
```

The default port where the service is located is 20504. Of course, The infrastructure can be installed also on a different port, by specifying the port value with the *-port* option. For instance, to install tucson on port 20505, type:

```
java -cp dir /tucson.jar alice.tucson.runtime.Node -port [20505]
```

Once the infrastructure is running, in order to launch `CLIAgent`, from a console command, type:

```
java -cp dir /tucson.jar alice.tucson.ide.CLIAgent
```

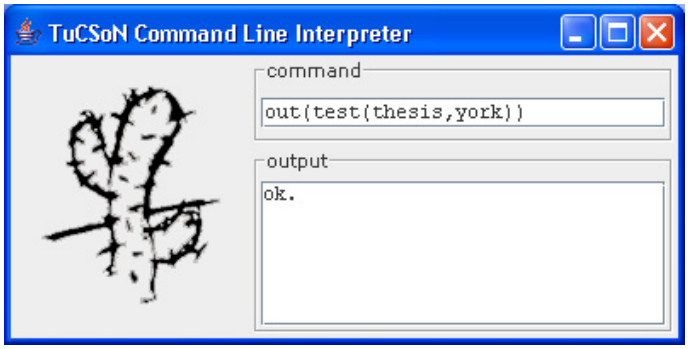

Figure 4.3: `CLIAgent` tool

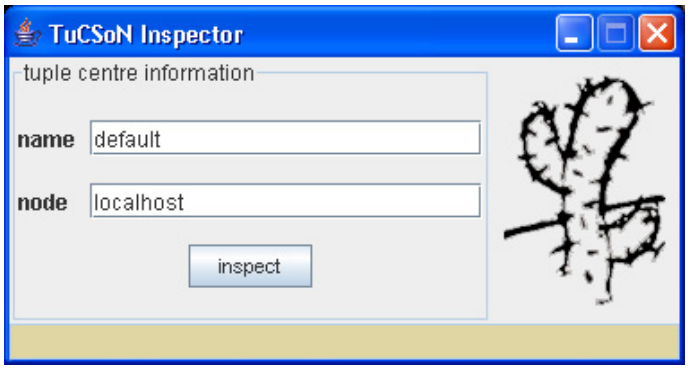

Figure 4.4: `Inspector` tool

and `CLIAgent` GUI should be appear as Fig.4.2.

Now, all possible commands shown in Tab.4.2 can be directly write in the *command* field. For more details see [11].

Another fundamental tool to monitor tuple centre communication and coordination state, and to debug tuple centre behaviour is `Inspector`.

The quickest way to launch `Inspector`, from a console command, is typing:

```
java -cp dir /tucson.jar alice.tucson.ide.Inspector
```

and `Inspector` GUI should be appear as Fig.4.3.

When tuple center name and node are specified, the 'inspect' button

allows to inspect:

- the tuple set
- the pending query set
- the triggered reaction set
- the behaviour specification set

`Inspector` also supports step-by-step tracing. For use and more details about this tool, see [11].

Virtually, it is possible to access into TuCSoN infrastructure with any hosting language. Currently, this is possible to do by means of *Java* and *Prolog*.

The fundamental packages, required by any *Java* program wishing to exploit the TuCSoN infrastructure, are `alice.tucson.api` and `alice.logictuple`. `alice.tucson.api` Whereas the former includes classes providing access to TuCSoN infrastructure, classes representing tuple centre identifier (`TupleCentreId`) and agent identifier (`AgentId`), the latter includes classes representing the communication language of TuCSoN, that is logic tuple.

The main classes that constitute the communication language are:

- *LogicTuple*: this class represents a logic tuple. Typically a logic tuple is characterised by a name and a list of logic arguments (TupleArgument);
- *TupleArgument*: this class represents the a tuple argument. It can be a value numbers, strings, structures, lists or a variable. This class provides all the services to manipulate tuple arguments, testing their type, converting them, retrieving their numeric / alphanumeric value. Derived classes Value and Var are provided for facilitating the construction of specific tuple arguments;
- *Value*: this class represents concrete data values of a logic argument. In particular Value objects can be created representing untyped values such as atoms (Strings) and structures (with a name and a set of arguments) and typed values such as numbers integers, long, floats, doubles.
- *Var*: this class represents logic variable, characterised by a name.

Instead, in order to access into TuCSoN infrastructure by means *Prolog*, the use of *tuProlog* is a solution. By writing the theory in a text file such as 'test.pl' and then by typing the following command, is possible execute a tuProlog agent:

```
java -cp tucson.jar alice.tuprolog.Agent test.pl main(localhost)
```

For more details in use of TuCSoN from Java and tuProlog, such as examples to create tuple centres, logic tuples and prolog agent using TuCSoN, 'TuCSoN Guide' [11] is suggested.

Finally, having in mind the fact that tuple centres are programmable tuple spaces, this mean that their behaviour can be programmed using a logic based language called ReSpecT, defining the coordination laws ruling agent interaction. More precisely, ReSpecT Reaction Specification Language creates the possibility to associate to a TuCSoN communication event (such as out, in, rd, inp, rdp) the specific activities of computation or the artifact actions called *reactions*. Every reaction can freely modify the data of the tuple centre, and to go off a communication event that can be intercepted by another reaction. A ReSpecT program takes the form of a set of reactions and roughly speaking, a specific ReSpecT tuple is a Prolog clause in the form:

```
reaction(Event, ( Body ))
```

where:

Event is an extended communication event, including the execution of basic coordination primitives (out, in, rd, inp, rdp), and the successful execution of some ReSpecT primitives (out_r, in_r, rd_r, no_r) described in Tab.4.3. *Body* is a sequence of ReSpecT predicates. The main predicates are listed in the Tab.4.3: basically, they make it possible to inspect and change the content of the tuple set, by inserting / retrieving / reading associatively tuples [11]. A simple example of reaction is:

```
reaction(out(p(X)),(
      out_r(backup(p(X))) )).
```

In this example, whenever a tuple matching the template p(X) is inserted in the tuple centre, another tuple backup(p(X)) is created.

MAIN ReSpecT PRIMITIVES	
out_r(T)	always succeed and as effect the logic tuple `T` is inserted in the tuple set
in_r(TT)	succeed if a logic tuple matching the template `TT` is present in the tuple set, and as effect the tuple is removed and unified with `TT`. Otherwise the predicate fails.
rd_r(TT)	succeed if a logic tuple matching the template `TT` is present in the tuple set, and as effect the tuple is unified with `TT`. Otherwise the predicate fails.
no_r(TT)	succeed if no logic tuple matching the template `TT` is present in the tuple set, otherwise the predicate fails.
pre	succeed if the event triggering the reaction is a request (`in` or `rd`) in the *pre* stage.
post	succeed if the event triggering the reaction is a request (`in` or `rd`) in the *post* stage.
success	succeed if the event triggering the reaction is a successful `inp` or `rdp`
failure	succeed if the event triggering the reaction is non successful inp or rdp
current_agent(TT)	succeed if `TT` unifies with a logic tuple denoting the identity of the agent responsible of the communication event which generated the reaction
current_tuple(TT)	succeed if `TT` unifies with current logic tuple specified in the event triggering the reaction
current_tc(TT)	succeed if `TT` unifies with a logic tuple denoting the name of current tuple centre
current_time(TT)	ssucceed if `TT` unifies with a logic tuple denoting current tuple centre virtual machine time

Table 4.3: ReSpecT primitives semantics [11].

These primitives are only the main ReSpecT primitives, but there are others different primitives such as `out tc(TC,T)` or `spawn(AgentID,AgentType,ArgList)`. For more details about these last primitives and how to program tuple centre behaviour, 'TuCSoN Guide' [11] is suggested.

Currently, the model for distributed programming is synchronous, based on directed point-to-point communication. The programming community appears to accept this without question. Although the Tuplespace model of communication has numerous benefits for the distributed application programmer, most Tuplespace efforts of the past remain unknown to the general population, having been known only to the relatively obscure area of parallel programming. However, with all those coordination languages pushing into the mainstream of computing, it is thought this is about to change.

In this chapter the main structures and properties of PyLinda, Lindacap and TuCSoN have been discussed in order to explain Role-Based models in these coordination languages. The next chapter will introduce some Role-Based models such as RBAC (Role-Based Access Control) model and Role-Based Workflow model.

Chapter 5

RBAC model in MAS

The goal of this thesis is to show and discuss the works developed by both the University of Bologna (Cesena) and the University of York, and in particular, a way to deepen Advanced Role-Based models. In the previous chapter Tuplespace languages from York & Cesena were discussed. This last chapter aims to satisfy the last part of the above goal introducing the RBAC (Role-Based Access Control) model and then comparing this model to ACL (Access Control Lists). Subsequently, RBAC in MAS will be shown in order to explain RBAC in TuCSoN. Moreover, CNP (Contract Net Protocol), as a case study, will be shown in Linda and in TuCSoN respectively. Finally, a possible way of bringing RBAC model in Lindacap will be explained.

5.1 Role-Based Access Control model

RBAC models and their extensions are currently considered the most effective approach for engineering access control in complex information systems and dynamic organisations [34]. Sandhu *et al.* discuss reference models for role-based access control (RBAC) [39], which originated with multi-user systems (mainframes etc) in the 1970s. Sandhu also concluded that although RBAC mechanisms show great promise, more work needed to be carried out to allow the specification and management of RBAC policies and constraints. The key concept behind RBAC is that roles are created for each job function, and permissions required to fulfil the job function are assigned to the role. This abstraction increases exibility, since both the permissions and the subjects associated with the role can change over time [24]. Mainly, in the basic level, Sandhu characterises RBAC systems into

four components:

1. a set of users
2. a set of of roles
3. a set of permissions
4. a set of sessions

In this model:

A *user* is a human being. The concepts of a user can be generalized to include intelligent autonomous agents such as robots, immobile computers, or even networks of computers.

A *role* is a job function or job title within the organization with some associated semantics regarding the authority and responsibility conferred on a member of the role.

A *permission* is an approval of a particular mode of access to one or more objects in the system. Permissions are always positive and confer the ability to the holder of the permission to perform some action(s) in the system. Some access control literature talks about 'negative permissions' which deny, rather than confer, access. The nature of a permission depends greatly on the implementation details of a system and the kind of system that it is.

A *session* is a mapping of one user to possibly many roles. A user can be a member of many roles, and a role can have many users. Similarly, a role can have many permissions, and the same permission can be assigned to many roles.

A short-coming of the base model is that there is no way to specify negative authorisations [24]. This is addressed in RBAC2 with the introduction of constraints. Constraints allow the specification and enforcement of restrictions where an individual is forbidden from taking on a particular role once he is already participating in a conflicting role. This is also known as the principle of separation of duties. By means of inheritance relations, one role can inherit permissions assigned to a different role. These inter-role relations can be used to enforce security policies that include *separation of duties* and *delegation of authority*. Separation of duties is achieved by ensuring that mutually exclusive roles must be invoked to complete a sensitive task, such as requiring an accounting clerk and account manager to participate in issuing a check. With separation of duty, RBAC directly supports other two well-known security principles:

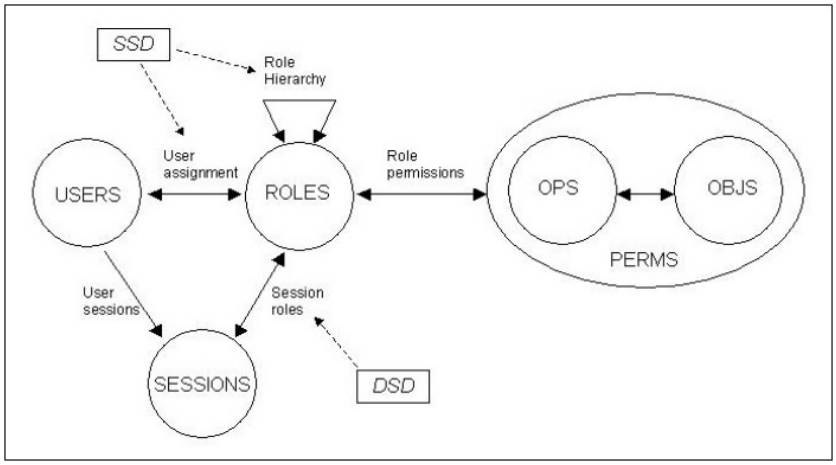

Figure 5.1: RBAC Reference Model [15, 34]

1. least privilege

2. data abstraction

Least privilege is supported because RBAC can be configured so that only those permissions required for the tasks conducted by members of the role are assigned to the role.

Data abstraction is supported by means of abstract permissions such as credit and debit for an account object, rather than the read, write, execute permissions typically provided by the operating system [34].

The key to RBAC lies in *user assignment* and *permission assignment* relations. Security policies are defined in terms of relationships between the element sets. *User assignment* relationships define which users are assigned to a specific role, which means that they are allowed to play it inside the organisation; *permission assignment* defines which permissions are assigned to each role. Thus, RBAC provides an encapsulation of security policy. Access control strategy is encapsulated in various components of RBAC such as role-permission, user-role and role-role relationships. These components, configurable dynamically by system administrators, collectively determine whether a particular user will be allowed to access a particular piece of data in the system. Moreover, the RBAC approach makes it possible & easy to incrementally evolve the access control policy over the system life cycle, to meet the changing needs of an organisation.

Finally, Fig.5.1 shown the components and relations of RBAC reference model, in which static separation of duty properties (SSD) are obtained by enforcing constraints on the assignment of users to roles; instead, dynamic separation of duty properties (DSD) are obtained by placing constraints on the roles that can be activated within or across a users sessions [34].

5.2 Comparing RBAC and ACL

The RBAC metaphor is powerful in its ability to express access control policy in terms of the way in which administrators view organizations. RBAC differs from access control lists (ACLs) used in traditional discretionary access control systems in that it assigns permissions to specific operations with meaning in the organization, rather than to low level data objects. For example, an access control list could be used to grant or deny write access to a particular system file, but it would not say in what ways that file could be changed. In an RBAC based system, the assignment of permission to perform a particular operation is meaningful, because the operations are fine grained and themselves have meaning within the application. RBAC is often distinguished from ACLs by the inclusion of a feature which allows a session to be associated with a proper subset of the roles (i.e. groups in ACL terms) authorized for a user [6]. RBAC has several advantages over ACLs. Even a very simple RBAC model affords an administrator the opportunity to express an access control policy in terms of the way that the organization is viewed. With RBAC, it is not necessary to translate a natural organizational view into another view in order to accommodate an access control mechanism.

For instance [6], to describe an access control policy using an ACL, where only groups are permitted as entries in the ACL, an administrator:

1. Creates groups of individuals according to their responsibilities.

2. Associates with these groups the permissions necessary for the individuals in the group to carry out their responsibilities.

To describe an access control policy using the minimal RBAC model (see RBAC0 in [39]), an administrator:

1. Creates roles based on the responsibilities necessary to meet the goals of the organization.

2. Associates these roles with the permissions necessary to carry out these roles.

3. Associates these roles to individuals.

ACL is equivalent to RBAC from the point of view that any access control policy described using ACL (in which only groups are permitted as entries in the ACL) can be described by the minimal RBAC model and any access control policy described by the minimal RBAC model can be described by this ACL. Barkley showed [6] that the ability to express access control policy using a very simple RBAC Model is no different from an ACL model, which is supported by many ACL mechanisms. Moreover, an RBAC mechanism consisting of a minimal RBAC and the role hierarchy, Static Separation of Duty, and Cardinality features of the NIST Model [13] can usually be implemented on a system that supports ACL. In addition, most RBAC models have features which most ACLs do not. In particular, some RBAC models [39] have role hierarchies where one role can inherit another.

5.3 RBAC in MAS

In contexts such as information systems, Role-Base Access Control (RBACs) models are currently considered the most effective way to engineer security for complex organisations [14, 39].

Their major properties concern the ability to articulate and enforce enterprise (system) specific security policies and to streamline the burdensome process of security management [14].

With respect to the previous approaches (discretionary and mandatory access control), they allow for more flexible and detailed control and management of security. Because of this flexibility, extensions have been layered on top of the basic RBAC model for effectively integrating security and organisation issues in the context of open, dynamic and complex systems, such as inter-organisational workflow management [23] and pervasive computing environments [41].

RBAC-like model can be suitably introduced in MAS models & infrastructures, integrating a high level role-based *security* (access control) approach with MAS coordination and *organisation*, where the role abstraction is already at play [34]. Conceiving an RBAC-like model into MAS shifts (or completes) the focus on the runtime aspects: roles, sessions, policies become runtime issues of a MAS organisation, manageable dynamically by suitable services provided by the infrastructure.

Omicini *et al.* [34] provide a solution based on the notion of Agent Coordination Context (ACC) in order to introduce an RBAC-like approach

in MAS (models / infrastructures) and to integrate it with possible MAS role-based models. In the following, this solution is shown.

The notion of ACC has been introduced in [28] as a way to model MAS environment from the individual agent viewpoint, more precisely the objective and subjective relationships between an agent and the (organisational) environment in which he is immersed [34]. The notion of context can be used as a first-class abstraction for modelling and engineering the environment. The full characterisation of ACC is obtained then by identifying the context abstraction as the conceptual place where to set the boundary between the agent and the environment, so as to encapsulate the interface that enables agent actions and perceptions inside the environment.

More precisely [34], an ACC:

1. works as a model for the agent environment, by describing the environment where an agent can interact;

2. enables and rules the interactions between the agent and the environment, by defining the space of the admissible agent interactions.

Two basic stages characterise the ACC dynamics: *ACC negotiation* and *ACC use*. An ACC is meant to be negotiated by the agents with the MAS infrastructure, in order to start a working session inside an organisation. The agent requests an ACC specifying which roles to activate. If the agent request is compatible with (current) organisation rules, a new ACC is created. The agent then can use the ACC to interact with the organisation environment, by exploiting the actions/perceptions enabled by the ACC [34]. In other worlds, ACCs have been introduced as a mean to model explicitly presence of an agent inside an (organisational) environment, both in terms of the actions/perceptions the agent can do (as a kind of environment interface) and their effects.

Summarizing the main concepts [34] of RBAC-like Model using ACC's:

- The ACC naturally embodies the concept of session, coupling dynamically agents (as users) and the organisation environment.

- ACCs represent the runtime entities that physically enable and constraint agent actions, according to rules which in the RBAC framework correspond to the role-permission relationships.

- Differently from the basic RBAC model, an agent can hold only one ACC at a time, but of course, the same ACC can involve the activation of multiple roles, and the same role can be played simultaneously in different ACC.

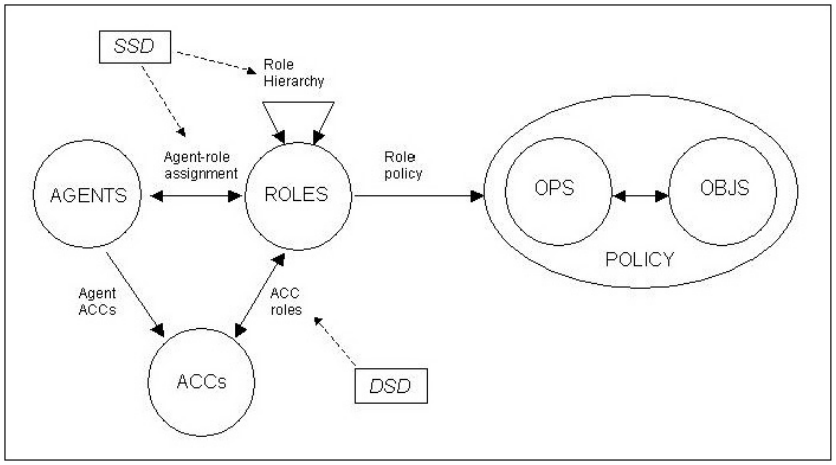

Figure 5.2: RBAC-like Model using ACC's [34]

Fig. 5.2 shows the use of the ACC for realising an RBAC architecture, in which agents are the users of the systems and the ACCs play the role of sessions. More precisely [30] the main peculiarities of RBAC-MAS with respect to RBAC, can be summarised as follows:

- Instead of RBAC users, RBAC-MAS has *agent classes* (`AGENTS`),
- Operations (`OPS`)and objects (`OBJS`) are not unrelated in MAS. Therefore RBAC-MAS introduces a notion of *action*, which is an operation over a given object.
- Instead of permissions, which are subsets of the cartesian product of the set of the operations and the set of objects, RBAC-MAS introduces policies (*POLICY*), which are protocols of actions.
- In the RBAC-MAS approach, the *ACC abstraction* (`ACCs`)also accounts for the RBAC notion of session. Since each agent is assigned its own ACC within an organisation, RBAC-MAS allows only one session per agent (per organisation).

ACC negotiation is ruled by both currently defined agent-role relationships (defining the SSD rules in Fig. 5.2) and inter-role relationships (used for defining DSD rules). Once the properly configured ACC has been released to the agent (after a successful negotiation), the relationships defined among roles and policies as depicted in Fig. 5.2 shape the agent action

space enabled by the ACC, defining what actions as operations within the environment the agent is allowed to execute.

Omicini *et al.* [30] show how an ACC can be used to:

- Enable protocols
- Enforce concurrency policies
- Measure and rule resource access
- Allow for dynamic access control.

In other words, the ACC is meant to play the role of infrastructure abstraction defining and encapsulating issues related to quality of service, and, in particular, of the communication between the agent owning the ACC and the organisation. So the ACC not only represents and enforces policy concerning agent role(s), but defines properties of interaction, established and tuned during ACC negotiation [34].

Finally, it is important to point out these further features [30] of RBAC-MAS:

- *Static / Dynamic Enforcement of Policies*: in the RBAC-MAS approach, policies are dynamically enforced via ACCs, which work as both design time and run time abstractions
- *Role Activation / Deactivation*: policies constrain which roles can be activated or deactivated by an agent given its current state of actions and active role sets
- *No Default Roleset*: RBAC-MAS does not provide a notion of default roleset to an agent.

5.4 RBAC in TuCSoN

In the previous section, the ACC abstraction has been exploited to bring an RBAC-like model in the TuCSoN coordination infrastructure: In TuCSoN the organisation resources accessed by agents are tuple centres (see section 4.3).

In Fig. 5.3 the RBAC-like architecture specialised in the TuCSoN case is shown.

More precisely [34], in this architecture:

- Tuple centres are the objects.

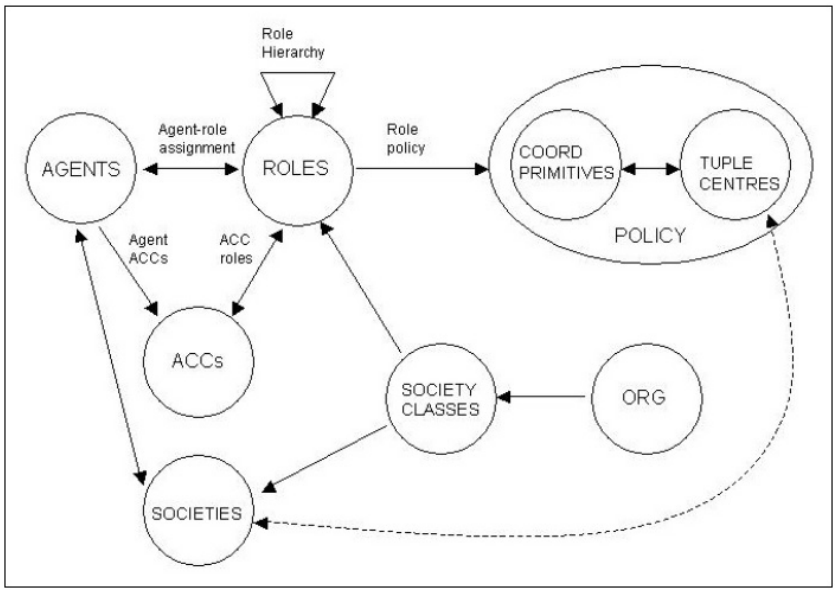

Figure 5.3: An RBAC-like Model using ACCs in TuCSoN [34]

- Tuple centre coordination primitives are the operations.
- A system is defined as an organisation (`ORG` in Fig. 5.3).
- An organisation is dynamically structured in societies.
- A society class groups a set of roles.
- Each role is characterised by a policy, defining the actions and interaction protocols allowed for agents playing such a role.
- An agent action has the form: `Tid @ Node ?  op` (see section 4.3).
- Permissions are expressed in terms of rules on allowed actions/interactions and their aggregation as protocols.
- Societies are dynamically composed by a set of agents and tuple centres.
- An agent takes part actively to an organisation by playing at least one role in a society.

In TuCSoN the *role policies* are described using a Prolog theory, as a set of rules defining what patterns of actions and interactions are allowed. The theory is composed by a set of `can_do` rules defining role action space [34, 31]:

`can_do(CurrentState,Action,NextState):- Conditions.`

where:

- The action `Action` can be executed in the role state `CurrentState` if conditions `Conditions` value, and in that case next role state is `NextState`.
- The concept of role state is used as a way to easily express interaction protocols;
- `Conditions` can contain also built-in predicates useful to describe context-aware (with respect to local time, space and identity/positions of the agents) policies.

The next sections will show CNP (Contract Net Protocol) as case study in Linda and TuCSoN, respectively.

5.5 CNP in Linda

The conventional CNP framework does not use a common data space for communication between the agents: the agents interact directly with each other. This point-to-point form of communication, however, does not use the flexibility inherent in open systems, like the Internet [42]. The features of the Tuplespace-based models, such as Linda [17] are desirable in coordinating the contracting process in the vastly distributed, heterogeneous, dynamic open system.

In this data-driven model [42]:

- the wide variety of agents is enabled;
- the agents involved do not need to be concerned with how the communication is done
- the associative matching rules allows the agents to only specify a pattern of the data they want, rather than explicitly specify the exact tuple

- the name-uncoupling, space-uncoupling and time-uncoupling properties gives a new dimension of contracting where the agents involved are not required to be known to each other, nor exist in the same place at the same time in order to communicate.

Briefly, in Tab.5.1, manager and contractor operations, in Contract Net Protocol, are shown.

Manager:

```
1  Send (broadcast) Task announcement to n agents
2  While (time > 0) or (numOfBids > 0)
3     Receive bids
4     Evaluate bids
5  Select Contractor
6  Send contract award to Contractor
7  Receive results
8  Synthesize results
```

Contractor:

```
1  Receive Task announcement
2  Evaluate my interest in Task
3  Send my bid to Manager
4  Receive answer to my bid
5  If my bid is accepted (awarded)
6     Perform Task
7     Send results to Manager
```

Table 5.1: pseudo-codes of agents in CNP [42]

Although Linda is proclaimed to be a powerful model for open communication system, it has its limitations, most notably its *insecurity*. As shown in Figure 5.4a, to 'broadcast' the task, the manager (agent A) simply drops the task announcement message (tuple `<t, ...>`) into the Tuplespace (TS), without having to specify any target recipients, and the potential contractors will listen to it. Once the tuple is deposited into TS, there is nothing to prevent any agent from 'accidentally' or 'maliciously' removing the tuple. Then, the interested agents (B and D) submit their bids to A by outing them into TS (Figure 5.4b). After retrieving and evaluating the bids, A decides to award the contract to B, thus putting the award tuple, `<aw, t,`

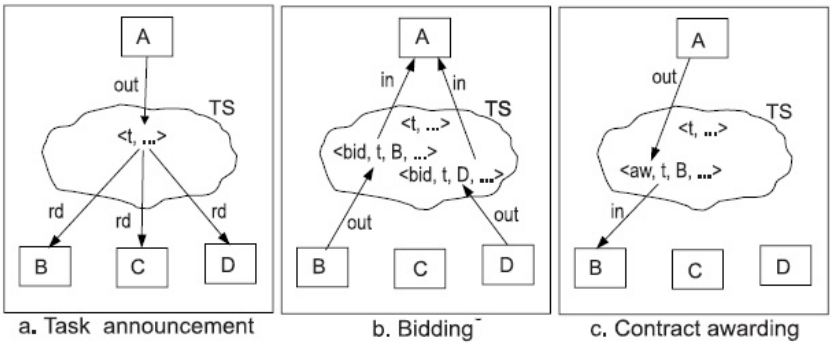

Figure 5.4: Implementation of CNP in Linda [42]

B> into TS. Unfortunately, as in the bid tuples, the award tuple is exposed to a third-party removal or the tuple can be forged by any other agent [42].

Many Linda variants introduce the idea of '*Multiple Tuplespaces*' that provide a private tuplesapce known only to the two agents concerned, in order to solve this security problem of Linda. However, this alternative is not practical in the open system as the reference to the private TS has to be passed (in a tuple) to the agents via a TS that can be globally accessed by both (and all) agents, thus exposing the reference tuple to other agents as well defeating the purpose of having a 'private' TS [42].

A solution is represented by attaching access control *attributes* [45] to objects (TSs and tuples). Controlling the access to TSs and tuples using attributes provides secure communication, in addition to the flexibility of the Linda model. The idea of using attributes [45] eliminates this problem: it enables one to specify the type of operation (write, read, or remove) allowed on tuples and TSs.

Another solution is represented by using *Scopes* [26]. Scope is a simple but powerful generalisation of Linda where the notion of TS is replaced with scope, 'a region of space which may contain or intersect with other scopes'. Without introducing new primitives for tuple access, Scope adds more fine grained control over the accessibility of tuples than is available in standard Linda by using scopes to limit tuples visibilities. Thus, Scopes properties allow one to perform operations not only on the tuples, but also on the region itself. These, along with its tagging and streaming abilities, offer a rich range of functionalities useful in developing various robust concurrent applications, primarily those that rely on secure communication in open system, such as CNP [42].

Finally, a simple version of CNP - see the next section - has been deployed in TuCSoN [12] which monitors the agents interaction (interaction protocols, coordination policies, and environment constraints) via its programmable tuple centres, expressed in the ReSpecT coordination rules.

5.6 CNP in TuCSoN

The role policies with a simple version of a well-known agent interaction protocol, the Contract Net (CNP) [40], are briefly illustrate. More precisely, in the following, CNP case study from Omicini *et al.* [34] is riposted. The hypothesis is to be in a task allocation scenario, where:

- `acme.org` is the name of organisation
- `task_distribution` is the name of society for task allocation coordination activity
- `master` and `worker` are two roles involved in the society.
- `tasks` is the name of tuple centre as coordination artifact used to support the task allocation coordination activity.

While the interactive behaviour of masters and workers is described in the pseudo-code in Tab.5.2 (top), the coordinating behaviour of the tuple centre is defined by the reactions shown in Tab.5.2 (bottom) expressed in the ReSpecT language. When a master issues a task announcement, reaction 2 is triggered and coordination data are set up in the tuple centre. Each time a worker makes a bid, reaction 1 stores information about the new proposal and updates the bid list. When the master eventually collects the bids, reaction 3 removes the task announcement tuple: so, no more bids are considered. Finally, when a master awards the contract, reaction 4 emits the tuple `bid(Task, TheBid, awarded)` to notify the specific worker waiting for that answer, then triggers reaction 5 and 6: the first places the `refuse_others` tuple, which is used to collect and remove the information about other bidders (tuples `contractor`), while reaction 6 notifies the other contractors by emitting the `bid(Task, TheBid, not-awarded)` tuple.

```
0  enterACC('acme.org',
1     [role(master,task_distribution)])
2  wait(ExpireTime)
3  tasks ?  in(bids(Task,BidList))
4  Bid ← selectWinner(BidList)
5  tasks ?  out(award(Task,Bid))
6  tasks ?  in(result(Task,Result)
```

```
0  enterACC('acme.org',
1     [role(worker,task_distribution])
2  MyBid ← evaluate(Task)
3  tasks ?  in(bid(Task,MyBid,Answer))
4  if (Answer=='awarded') {
5     Result ← perform(Task)
6     tasks ?  out(result(Task,Result)}
```

```
1  reaction(in(bid(Task,MyBid,Answer)), (
     pre,out_r(contractor(Task, MyBid)),
     in_r(bids(Task, L)),
     out_r(bids(Task, [MyBid|L])) )).

2  reaction( out(announcement(Task) ), (
     out_r(bids(Task,[])) )).

3  reaction( in(bids(Task,L) ), ( post,
     in_r(announcement(Task)) )).

4  reaction( out(award(Task,TheBid) ), (
     in_r(award(Task,TheBid)),
     in_r(contractor(Task,TheBid)),
     out_r(bid(Task,TheBid,awarded)),
     out_r(refuse_others(Task)) )).
```

```
5  reaction(out_r(refuse_others(Task)), (
     in_r(refuse_others(Task)),
     in_r(contractor(Task,TheBid)),
     out_r(bid(Task,TheBid,'not-awarded')),
     out_r(refuse_others(Task)) )).

8  reaction(out_r(refuse_others(Task)), (
     in_r(refuse_others(Task)),
     no_r(contractor(_,_)) )).
```

Table 5.2: (*Top*) The behaviour of master (left) and worker (right) agents in the CNP example; (*Bottom*) The ReSpecT code defining the behaviour of the tuple centre tasks, implementing the coordination rules of the Contract Net Protocol.

In Tab.5.3 (*top*) is shown a possible role policy for the *master* role, specified by the society administrator and enacted by the ACC. An interaction protocol, composed by four different role states (`init, task_announced, choose_bidder, get_result`) is enforced by the policy. In Tab.5.3 (*bottom*) is shown a possible role policy for the *bidder*. Also for the bidder a protocol composed basically by two states (`init, awarded`) is enforced by the policy.

```
can_do(init, tasks ?  out(announcement(Task)), task announced(Task)).
can_do(task_announced(Task), tasks ?  rd(announcement(Task)),_).
can_do(task_announced(Task), tasks ?  rd(bids(Task),_)),_).
can_do(task_announced(Task), tasks ?  in(bids(Task),BidList)),
                             choose_bidder(Task,BidList)).
can_do(choose_bidder(Task,BidList), tasks ?  out(award(Task,Bid)),
                             get_result(Task,Bid)):- element(Bid,BidList).
can_do(get_result(Task,_)), tasks ?  rd(result(Task,_)),_).
can_do(get_result(Task,_)), tasks ?  in(result(Task,_)),init).
```

```
can_do(init, tasks ?  rd(announcement(_)),_).
can_do(init, tasks ?  in(task(Task,_,awarded)),awarded(Task)).
can_do(awarded(Task), tasks ?  out(result(Task,_),init)).
```

Table 5.3: Role policy for the master role (*Top*) and worker role (*Bottom*) [34].

5.7 Possible RBAC in Lindacap

The main feature of Lindacap (see section 4.2) is its *multicapabilities* which generalize capabilities to collections of objects. The concept of capability may provides the best-fit solution to overcome the problems caused by the loosely controlled coordination of tuple-space systems [43]. This concept may be used, not only into this purpose, but also to see a capability (or a multicapability) as an entity that play the role of permission in the RBAC model. More precisely, in Fig.5.5, a possible way to bring RBAC model in Lindacap is shown.

The main peculiarities of this RBAC model compared to the original RBAC model [39] can be summarised as follows:

- Agent Classes: obviously, instead of users, RBAC in Lindacap has agent classes such as RBAC in MAS [see section 5.3].
- Capabilities and Multicapabilities: instead of permissions - that are approvals of a particular mode of access to one or more objects in the system - RBAC in Lindacap introduces capabilities. Infact a capability [20] specifies which kind of actions on a certain object are permitted to the holder of the capability. Instead, multicapability extends capability for a collection of objects.

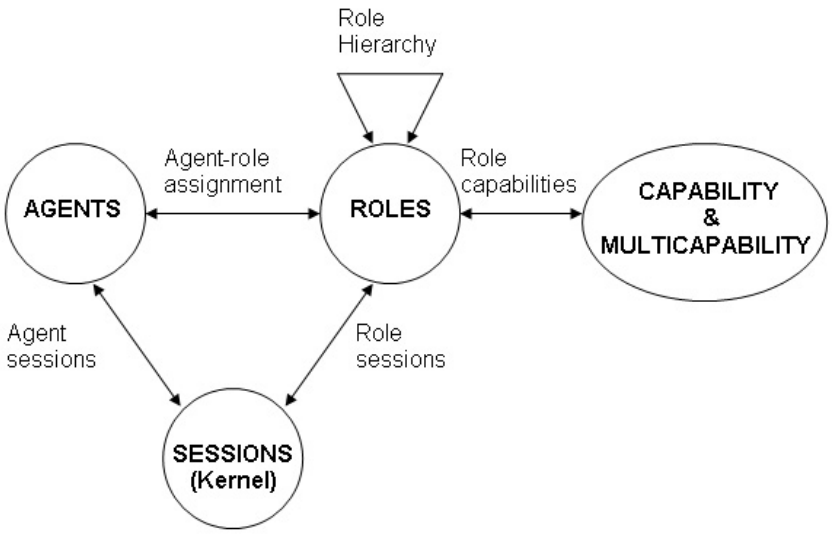

Figure 5.5: Possible RBAC model in Lindacap

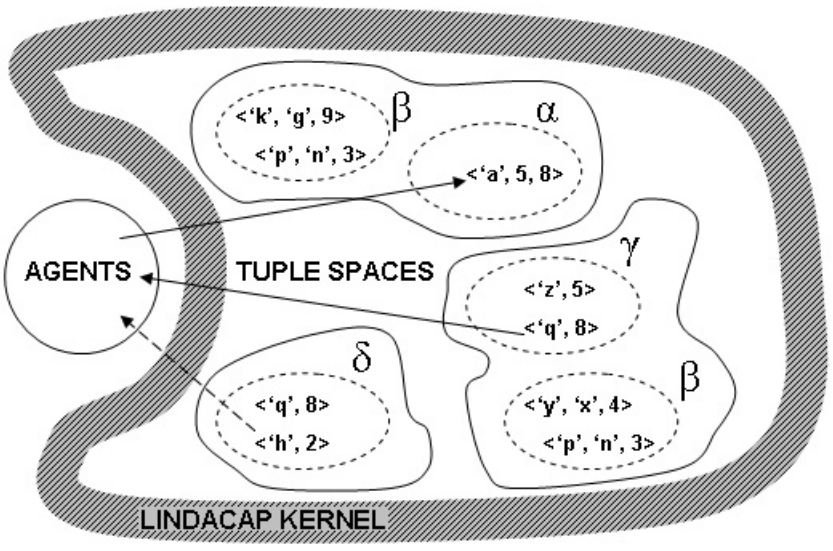

Figure 5.6: Kernel in Lindacap

- Role: in this case, a role - that usually represent a job function or job title within the organization - can be seen as an abstraction that may englobes more than one multicapability.

- Session(Kernel): the basic session concept is the same of the original RBAC model. In other words, a session is a mapping of one user(agent) to possibly many roles but, in the RBAC model in Lindacap, it can be seen such as the Lindacap kernel as well. More precisely, the term 'kernel' refers to the underlying distributed coordination mechanism that controls all operations in the system. It represents the totality of the Linda 'middleware'. Before referring to a class of tuples, either for output or retrieval, an agent must first request an appropriate multicapability (that may represent a part of a role) from the kernel. Fig.5.6 shows that any agent have to go through the Linda Kernel to make any operations to/from tuple space in order to recive an appropriate multicapability (such as α, β and γ) from the kernel. In this sense the Lindacap kernel can be seen as a mapping of agents to possibly many roles.

In this last chapter Role-Based Access Control models have been shown and discussed. More precisely, the general model of RBAC, a comparison between this model and ACL and an introduction of RBAC in MAS (by the ACC notion) have been discussed. Subsequently, RBAC in TuCSoN and CNP case study in Linda and in TuCSoN have been respectively shown. Finally, a possible way of bringing the RBAC model into Lindacap has been explained.

Conclusions

In this thesis the works about both the University of Bologna (Cesena) and the University of York are shown and discussed. More precisely, in order to explore tuplespace languages, agents and MAS overview and general concepts of Coordination Languages have been introduced first.

Subsequently, while some general tuplespaces languages (such as BeLinda, WCL, JavaSpaces and TSpaces) have been shown, the tuplespace languages developed by both Universities (such as TuCSoN, PyLinda and Lindacap), have been discussed in more details.

Finally the general model of RBAC, a comparison between this model and ACL, an introduction of RBAC in MAS (by the ACC notion), RBAC in TuCSoN and a famous case study (CNP) in Linda and in TuCSoN have been discussed.

An RBAC-like approach makes possible to gain all the benefits of the approach in engineering security inside complex MAS organisations, mainly in terms of encapsulation of the security policies, and flexibility in their management. In order to bring RBAC model in TuCSoN, ACCs have been the key for porting the model on top of the existing infrastructure, integrating coordination, organisation and security in a coherent way. The resulting security & organisation model provides *features* that are essential for the engineering of open MAS such as *dynamism/flexibility*, *support for heterogeneity* and *formal property verification* [34].

In the end of this thesis, in order to combine some different works achieved by both Universities, also a possible way of bringing the RBAC model into Lindacap has been discussed.

This thesis aimed to show and discuss the works on coordination languages and Role-Based models developed at the University of Bologna (Italy) and at the University of York (UK). Thus, the goal of this thesis has been successfully reached.

Bibliography

[1] G. A. Agha and R. Ziaei. Security and fault-tolerance in distributed systems: An actor-based approach. In *CSDA '98: Proceedings of the Conference on Computer Security, Dependability, and Assurance*, page 72, Washington, DC, USA, 1998. IEEE Computer Society.

[2] B. Angerer. Space-based programming. `www.onjava.com`, Mar. 2003.

[3] F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061, pages 34–56, Cesena, Italy, 1996. Springer-Verlag, Berlin.

[4] F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.

[5] S. C. Associates. Linda user guide, Sept. 2005.

[6] J. Barkley. Comparing simple role based access control models and access control lists. In *RBAC '97: Proceedings of the second ACM workshop on Role-based access control*, pages 127–132, New York, NY, USA, 1997. ACM Press.

[7] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[8] P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2), June 1996.

[9] P. Ciancarini. Coordination models, languages, architectures and applications: a personal perspective. Technical report, University of Leuven, February 1997.

[10] R. P. de Menezes. *Resource Management in Open Tuple Space Systems*. PhD thesis, Department of Computer Science, University of York, 1999.

[11] DEIS, U. degli Studi di Bologna, and Italy. TuCSoN guide, Dec. 2004.

[12] E. Denti, A. Omicini, and A. Ricci. Coordination tools for the development of agent-based systems. In J. Müller and P. Petta, editors, *Proceedings of the Third International Symposium "From Agent Theory to Agent Implementation"*, 2002. located at the 16th European Meeting on Cybernetics and Systems Research (EMCSR 2002).

[13] D. Ferraiolo, J. Cugini, and R. Kuhn. Role-based access control: Features and motivations. In *Proceedings of the 11th Annual Computer Security Applications Conference*. IEEE Computer Society Press, 1995.

[14] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[15] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[16] E. Freeman and S. Hupfer. Make room for javaspaces, part 1. *JavaWorld*, November 1999.

[17] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1), January 1985.

[18] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2), Feb. 1992.

[19] Hadeli, P. Valckenaers, M. Kollingbaum, and H. V. Brussel. Multi-agent coordination and control using stigmergy. *Comput. Ind.*, 53(1):75–96, 2004.

[20] G. J. Nutt. *Operating System: A Modern Perspective (2nd)*. 2nd edition, Addison-Wesley, 2002.

[21] J. Jacob and A. Wood. A principled semantics for `inp`. *Coordination Models and Languages*, 1906, 2000.

[22] S. Kambhatla, J. Inouye, and J. Walpole. Experiences with belinda: a syntethic linda benchmark for parallel computing platforms. *Proceedings of the 1990 International Conference on Parallel Processing*, 2, August 1990.

[23] M. H. Kang, J. S. Park, and J. N. Froscher. Access control mechanisms for inter-organizational workflow. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 66–74, New York, NY, USA, 2001. ACM Press.

[24] J. Knottenbelt. Policies for agent systems. Master's thesis, Imperial College London, 2001.

[25] V. Lesser. Reflections on the nature of multi-agent coordination and its implications for an agent architecture. *Autonomous Agents and Multi-Agent Systems*, 1, 1998.

[26] I. Merrick and A. Wood. Scoped coordination in open distributed systems. *Proceedings of the 4th International Conferance on Coordination Languages and Models*, pages 311–316, 2000.

[27] H. S. Nwana and D. T. Ndumu. An introduction to agent technology. *Re-Drawn by Mobile Computing*, 2003.

[28] A. Omicini. Towards a notion of agent coordination context. In D. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*, pages 187–200. CRC Press, 2002.

[29] A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, Nov. 2001.

[30] A. Omicini, A. Ricci, and M. Viroli. An algebraic approach for modelling organisation, roles and contexts in MAS. *Applicable Algebra in Engineering, Communication and Computing*, 16(2-3):151–178, Aug. 2005. Special Issue: Process Algebras and Multi-Agent Systems.

[31] A. Omicini, A. Ricci, and M. Viroli. RBAC for organisation and security in an agent coordination infrastructure. *Electronic Notes in Theoretical Computer Science*, 128(5):65–85, 3 May 2005. 2nd International Workshop on Security Issues in Coordination Models, Languages and Systems (SecCo'04), 30 Aug. 2004. Proceedings.

[32] G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46:329–400, 1998.

[33] A. Ricci. *Engineering Agent Societies with Coordination Artifacts and Supporting Infrastructures.* PhD thesis, Dipartimento di Elettronica, Informatica e Sistemistica, Universit degli Studi di Bologna, 2004.

[34] A. Ricci, M. Viroli, and A. Omicini. An RBAC approach for securing access control in a MAS coordination infrastructure. In M. Barley, F. Massacci, H. Mouratidis, and P. Scerri, editors, *1st International Workshop "Safety and Security in MultiAgent Systems" (SASEMAS 2004)*, pages 110–124, AAMAS 2004, New York, USA, 20 July 2004. Proceedings.

[35] A. Rowstron. WCL: A coordination language for geographically distributed agents. *World Wide Web Journal*, 1(3), 1998.

[36] A. Rowstron and A. Wood. Bonita: A set of tuple space primitives for distributed coordination. *Proc. 30th Hawaii International Conference on System Sciences*, 1:379–388, 1997.

[37] A. Rowstron and A. Wood. Solving the linda multiple `rd` problem using the `copy-collect` primitive. *Science of Computer Programming*, 31(2–3):335–358, 1998.

[38] S. Russell and P. Norvig. *Solution Manual for Artificial Intelligence: A Modern Approach.* 2nd edition, Prentice Hall, 2003.

[39] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[40] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In *Proceedings of the 1st International Conference on Distributed Computing Systems*, pages 186–192, Washington D.C., 1979. IEEE Computer Society.

[41] A. Tripathi, T. Ahmed, D. Kulkarni, R. Kumar, and K. Kashiramka. Context-based secure resource access in pervasive computing environments. *percomw*, 00:159, 2004.

[42] N. I. Udzir and A. Wood. Implementing contract net in tuple space models. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2004)*, pages 325–330, Innsbruck, Austria, 2004. ACTA Press.

[43] N. I. Udzir, A. M. Wood, and J. L. Jacob. Coordination with multi-capabilities. In *COORDINATION*, pages 79–93, 2005.

[44] A. Wilkinson. Pylinda - distributed computing made easy. `http://www-users.cs.york.ac.uk/~aw/`, 2004.

[45] A. Wood. Coordination with Attributes. In P. Ciancarini and A. Wolf, editors, *Proc. 3rd Int. Conf. on Coordination Models and Languages*, volume 1594, pages 21–36, Amsterdam, Netherland, 1999. Springer-Verlag, Berlin.

[46] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Journal of Research and Development*, 37(3 - Java Techonology):454–474, 1998.

Acknowledgments

I would want to thank all my Teachers: Andrea Omicini that was always available to give me good suggestions, Alan Wood for his willingness to help and his kind contributions, Alessandro Ricci and Mirko Viroli.

A great thanks to my family: to my parents Alberto and Giovanna, to my brothers Fabrizio and Silvia for the help, affection and the support necessary, to my girlfriend Angela that always encouraged me and helped me during the whole of this difficult period.

Finally, a big thankyou to all my friends and all people that helped me.