APROCO
A Programmable Coordination Medium

Diploma Thesis

of the Faculty of Sciences
University of Berne

by

Daniel Kühni

October, 1998

Supervisors:
Juan Carlos Cruz
Sander Tichelaar
Prof. Dr. Oscar Nierstrasz
Institute of Computer Science and Applied Mathematics

Typesetting done with much endurance using LaTeX and its companions.

Author's address:

Software Composition Group
University of Berne
Institute of Computer Science and Applied Mathematics (IAM)
Neubrückstrasse 10
CH-3012 Bern
Switzerland
Email: dkuehni@iam.unibe.ch or daniel.kuehni@computer.org
WWW: http://www.iam.unibe.ch/~scg/

Abstract

To keep up with rapidly changing requirements applications are increasingly built out of software components. A new trend is now to give those software components control over their own actions, to turn them into concurrently running software agents. These software agents have to be relatively independent to keep them exchangeable. Although independent, they still need to interact in order to achieve the application's overall goal. This results in the need to coordinate their interactions.

A number of coordination models were created to express common coordination solutions. Linda is one of the most prominent representatives of such coordination models. Linda is widely used because it offers simple means to separate coordination code from computational code within a single agent. Linda also offers a high degree of decoupling of agents through its generative communication style. However, Linda offers no direct support for the concentration of the coordination aspects of a whole application in a single location. Furthermore, Linda only offers a set of primitive operations and leaves the user with the task to construct realistic coordination abstractions out of them.

Coordination abstractions are often hard-coded into the participant agents' protocols and therefore neither flexible nor reusable. They are typically spread all over the application and it is almost impossible to identify them. It is not easy to encapsulate coordination abstractions because coordination typically affects multiple agents, and in open systems other requirements, such as flexibility and security, must also be dealt with.

We propose an open, flexible and extensible architecture for explicit coordination abstractions in open systems, called APROCO.

Our solution is based on the insight that separation of concerns (coordination and computation) is a necessary precondition for building reusable parts. The client agents of APROCO communicate through shared data spaces known from Linda using its generative communication style. The coordination between the participating agents is managed through special coordination agents that implement the used coordination abstractions. We present a list of coordination abstractions in open systems and show the applicability of the approach with some examples.

Acknowledgments

First and foremost I want to thank my supervisor **Juan Carlos Cruz** for his endurance in supporting me, discussing the numerous problems and obstacles I ran into, and providing me with tons of scientific papers to read and incorporate in this thesis. He also deserves thanks for tossing me into the art of graphology of ancient Egyptian hieroglyphs, at least this is what his comments on draft versions of this thesis looked to me.

To the same extend I want to thank my second supervisor **Sander Tichelaar** for his inspiring ideas and feedback and for invaluable remarks on several drafts of this thesis. He also never stopped to encourage me when I couldn't see the light at the end of the tunnel.

I want to thank Dr. **Stéphane Ducasse** for his comments and critical remarks on drafts of this thesis.

Writing my thesis within the Software Composition Group (SCG), lead by Prof. Dr. **Oscar Nierstrasz**, was a great experience for me. I enjoyed the friendly atmosphere, the inspiring discussions, and the interesting presentations from group members and visitors from all over the world. Thanks to the whole group for sharing your ideas and occasionally cakes and cookies with me.
Thanks go to the whole group, but especially to:

- Franz for the good teamwork during my time as teaching assistant in the programming languages lecture and his help to build up a nice LaTeX environment for the Java code in this thesis.
- Matthias for occasional concoctions of dark fluids considered as coffee.

Furthermore I want to thank my fellow students:

- Fredi for introducing me to the political landscape of central Switzerland.
- Manuel for mentioning me in the acknowledgment of his master's thesis.
- Michael for sharing the pleasure of having lunch in the mensa.
- Michele for his courageous fight for our ZIP drive.
- Tobias for philosophic discussions that prevented me from taking my work too serious.

A very special thanks goes to my best friend and wife Sarah for her enduring support and understanding throughout the ups and downs associated with a master's thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is widely accepted today that we cannot keep up with the pace of the changing user requirements with closed and proprietary software systems anymore. Most modern software applications are built out of ideally small, manageable pieces of software, possibly originating from different sources; the software components. Components are considered to be the best solution at hand to tackle the evolution of requirements, because when the requirements change, at least in theory, only the affected components need to be unplugged and reconfigured [ND95]. A current trend is to give those software components control over their own actions, to turn them into software agents. Software agents run concurrently on a computer system and have to communicate with each other in order to achieve a common goal. Because the software agents run concurrently and access resources concurrently, they not only have to communicate, they have to coordinate their actions in order to achieve the common goal. If we can divide an agent's behavior into a computation and a coordination part, this gives us the needed separation of concerns to be able to focus on a particular aspect. This separation is hard to achieve, because the coordination code is usually intermixed with the computational code. Furthermore, the coordination code is often hard-coded into the agents themselves and therefore neither adaptable to changing requirements nor reusable in different settings. To achieve reusability and flexibility we need to factor out the coordination code and make it explicit.

The idea of factoring out coordination code is all but new, Toby Bloom [Blo79] pointed out already in 1979 the need to factor out synchronization code to achieve ease of use and modifiability. As a more recent area of research aspect-oriented programming (AOP) [KLM+97] allows different aspects of an application (coordination is one of these aspects) to be expressed in a special programming language separated from the computational parts of the application. If we encapsulate this coordination code into an explicit entity that can be modified or exchanged, we achieve the desired flexibility. To achieve the desired reusability it is necessary to find common coordination solutions and abstract them from its concrete participants. It has been shown in the domain of object-oriented software construction that explicit connector objects are well-suited for coordinating active objects that communicate via message-passing [Gün98]. We take an agent to be a more general concept than an active object. Following [Hol97] we take an agent to be an abstraction for modeling concurrent entities, while the term active object is used to emphasize the view of an entity as encapsulating an active thread of control and thus denotes a possible implementation of

an agent. Because of this broader concept of agents, we used software agents as the active entities to study the coordination problems with their interaction.

In this thesis we describe an architecture to express and study coordination abstractions in agent-based open systems. The coordination abstractions are encapsulated into agents themselves to provide flexibility and reusability. The agents communicate using the so called *generative communication style* [Gel85] known from the coordination language Linda [CG90]. In the generative communication style, agents communicate through a shared data storage by producing and consuming data items. The resulting decoupling of sender and receiver is an important property in open systems.

1.1 Context

1.1.1 Open Systems

Open systems are application systems that are open in terms of topology, platform and evolution of requirements [Tsi89]. Open systems are best suited to keep up with the changes we experienced in the use of computers in the last decades: from huge, monolithic and proprietary mainframe solutions towards thin, networked, configurable and plug-extendable applications. The size and complexity of the applications has almost exploded, but the need for thinner applications, that can be adapted to the user's needs has been widely recognized to be a major goal of the years to come.

To be able to keep up with changing requirements, applications are best built out of components, because changes can then be coped with by reconfiguring or exchanging only the affected components instead of the whole application [ND95]. The software industry is developing component models (e.g. JavaBeans^{TM}[Ham97] from Sun or COM(+) [Rog97] from Microsoft) and integration techniques to allow us to build components and to assemble them into applications.

As an aspect of openness in topology, open systems can be seen as evolving systems of interacting entities ("objects", "agents", or "actors"). This implies that those entities can join or leave the actual setting of the system at any time.

1.1.2 Separation of Concerns

It has proven fruitful in many areas to split complex problems down to simpler problems that are more accessible to our inherently limited minds. This is especially true for the domain of software development, where today's applications are of remarkable complexity. The best way to divide complex problems is along orthogonal aspects, meaning that they are as independent as possible from each other. Therefore it is important to identify such orthogonal aspects.

In the domain of coordination as described in the next section the starting point of all research is to consider coordination as an orthogonal aspect to computation [CG90].

program = computation + coordination

1.1.3 Coordination

Coordination research has recently attracted a lot of attention in various scientific communities (see the next chapter for an overview of widely used definitions). In the area of open systems,

coordination has to do with the management of the communication and synchronization between concurrent entities. Some results from the research in coordination are various coordination models and languages such as Linda [CG90], ActorSpace [CA94] or Gamma [BLM93], just to mention a few (see e.g. [Cia97] or [PA98] for a survey). Coordination models based on generative communication [Gel85] are considered well-suited for the domain of open systems [Hol97]. Generative communication is based on a *shared data space* for the agents to communicate and synchronize with each other. With generative communication the sender generates data items and stores them in the shared data space. The receiver can then inspect the shared data space for specific data items and read or consume them. Generative communication offers a good basis for open systems support, because it uncouples communicating agents. This means that introducing new agents to a system does not lead to the necessity to identify them to all the existing agents to allow them to communicate with each other as would be necessary using message-passing. Furthermore, agents that leave the system do not leave other agents with invalid references to them.

1.2 Problem Description

Coordination research is a new way of understanding the complex interaction patterns that results from a potentially high number of concurrently running active entities constituting a single application. The long-term goal of the activities in this scientific community is to find simple ways to express and reuse solutions to common coordination problems. In the domain of open systems the flexibility of the coordination solutions is of major importance [Tic97]. Flexibility is understood as the property to be able to cope with openness in topology, platform, and user requirements.

To achieve flexibility and reusability in the coordination solutions it is recognized that the coordination part of the active entities needs to be separated from the computational part. This can be partly done using a coordination language such as Linda, which offers a small set of simple primitives that can be added to standard programming languages to express the coordination actions taken by an active entity. However, this separates coordination from computation only on the level of single program statements. The coordination code is still intermixed with the computational code. For a clean separation of concerns it is necessary to be able to separate all coordination code into explicit entities. This separation can also be seen in the domain of software architecture design, where an application is split up into *components* and *connectors* [GS94]. Connectors represent design decisions concerning the collaboration of software components. Specification languages for such connectors can help to describe designs that have specific properties such as component substitutability. Such design properties can only be safely transfered into the implementation if the programming language supports appropriate constructs for connectors.

We want to investigate Linda's properties in conjunction with the capabilities of agents as the encapsulation of coordination solutions. Linda is small and simple and offers some useful properties in the domain of open systems, but pure Linda has also some disadvantages: (i) Linda is not a concrete language, it is rather a set of "add on" primitives, which results in a very limited expressive power. One still has to program realistic coordination abstractions on top of the ones offered by the model. (ii) The coordination code is often intermixed with the computational code within a component. (iii) The coordination solution is spread all over the participating components. This prevents one from successfully reusing the coordination solution in different settings.

Furthermore, an adaption of the coordination behavior to changed requirements is not feasible, because it needs the adaption of every single component.

1.3 APROCO: A Programmable Coordination Medium

We propose that to be able to reuse a coordination solution it is necessary to abstract it from the coordinated agents and to encapsulate it into an explicit entity, a *coordination agent*. We want to use agents to encapsulate the coordination abstractions, because agents allow us to describe coordination services that are not merely reactive. This means, the client agents do not have to initiate these services in order to turn them operational.

We propose that by combining coordination agents into a *coordination medium* that delivers all the desired coordination services to its client agents, it is possible to take the coordination aspects out of the client agents' protocols and make them cleaner and easier to understand and reuse as well. A coordination medium is the means to enable communication between the agents and serves to aggregate agents into an ensemble. The coordination medium, APROCO, that we describe in this thesis is more than a coordination medium as just defined; it offers coordination services to its client agents that can be exchanged and modified and is thus a programmable coordination medium [DNO97] as we will show in chapter 3.

In this thesis we present the architecture of APROCO. APROCO is based on generative communication and uses the standard Linda operations to allow the client agents to communicate with each other. The information exchanged between the client agents is wrapped into *forms* and stored in shared data spaces. A form is a set of bindings of labels with corresponding values. The idea of agents communicating using forms is taken from [LAN98]. We used Linda-like operations (working on forms instead of tuples, as we describe in detail later) as the basic coordination level in APROCO because of their simplicity and easy implementation in usual programming languages. To overcome the limitations of Linda we encapsulated the coordination solutions into explicit entities, the coordination agents. They are coordinating the activities of the client agents connected to the coordination medium using the same basic Linda operations on shared data spaces to inspect and possibly transform the flow of information between the client agents. We present a list of coordination agents that we found to be useful in sample applications.

We are using Linda as the basis for APROCO because of its decoupling of senders and potential receivers of data. Linda is the "coordination assembler" that we use to build our agent-based coordination medium APROCO upon.

Architecture. We describe an architecture for the study of coordination problems with software agents in open systems. The architecture consists of client agents and a coordination medium that offers coordination services to the clients.

Coordination Agents. The coordination services provided to the clients of the coordination medium are implemented in coordination agents. The coordination code is encapsulated in an explicit entity that can be replaced or changed and even reused in different settings, because it is abstracted away from the real participants. The encapsulation of the coordination services into agents yields flexibility as well as the possibility of defining pro-active services, i.e. services

that are operational without being activated by a client. If all agents are using only generative communication, we found that we need multiple data spaces to allow the coordination agents to control the flow of information between the client agents.

List of Useful Coordination Abstractions. We present a list of useful coordination abstractions encapsulated into coordination agents that can be used within APROCO. We took an experimental approach to find these coordination abstractions by investigating small but typical real-world applications. The presented list of coordination agents is not complete and can be extended as needed. It can be found in section 3.5.

Needed Properties of a Coordination Medium for Agents. While constructing APROCO we found a list of properties that a coordination medium for agents in open systems must provide.

- **Clean separation of concerns.** To address the problem of reusable and flexible coordination solutions, a clean separation of coordination aspects from the computational aspects of an application is a prerequisite.
- **Explicit representation of the coordination solution.** The possibility to express a coordination solution in an explicit entity addresses the problem of easy flexibility. The coordination solution can be adapted by only changing one single component, the coordination agent.
- **Dynamic re-composition.** The possibility to dynamically join or leave a configuration addresses the problem of openness. The agents need to be as loosely coupled as possible (as with generative communication) to enable composition and re-composition as well as the clean encapsulation of the coordination solutions.
- **Access rights on data.** Access rights on data addresses the problem of secure communication between participating agents.

Implementation. The coordination medium APROCO and the examples have been implemented in Java using the Linda implementation Jada [CR96] for the implementation of shared data spaces and the basic Linda operations. All the material presented in this thesis is freely available at the author's web page (URL: `http://www.iam.unibe.ch/~dkuehni/aproco.html`).

1.4 Structure of the Thesis

The thesis is organized as follows:

The next chapter defines some basic notions and concepts we use throughout this thesis and shows to which aspects we will constrain our work. It also shows the limits of standard Linda to directly express reusable high-level coordination abstractions.

In chapter 3 we present the architecture APROCO. APROCO is based on generative communication and uses the standard Linda operations to allow the client agents to communicate with each other. The information exchanged between the client agents is wrapped into forms and stored in

shared data spaces. The coordination between the client agents is performed by the coordination agents inside the coordination medium. The coordination agents encapsulate solutions for specific coordination problems, abstracted from the real participants. We present a list of coordination agents that we found to be useful in sample applications. We give justifications for the design decision we made and conclude the chapter with an evaluation of our approach.

In chapter 4 we present an overview of the implementation of APROCO using Java and a Linda implementation, called Jada ("Java Linda").

Chapter 5 demonstrates the benefits of APROCO using a small set of sample applications. We show the coordination agents in actual configurations and describe their functionality in more detail. We discuss the properties of APROCO that are used in the examples and evaluate the solutions.

Chapter 6 summarizes the main contributions of the thesis and outlines areas for further research.

Chapter 2

Problem Domain

In this chapter we describe the problem domain of coordination in open systems. We define what we mean with often and differently used (and sometimes abused) terms such as “agent” and “coordination” and show to which aspects we constrain this work. We show an example of a software systems design and show the necessity of encapsulating the coordination code into a separate entity to gain flexibility and reusability. The decoupling of the software agents is an important property of a coordination medium suited for open systems that evolve over time. We present the coordination model Linda and its generative communication style that delivers the means for decoupling of agents necessary in open systems. We present an example Linda program and show its simplicity to express low-level coordination abstractions as well as its limits in terms of flexibility and reusability.

2.1 Agents

There are probably as many definitions of the term agent as there are people interested in this topic. A good survey on agent definitions can be found in [FG96]. The essence of all agent definitions is the following definition that we adopt for the agents used in this thesis.

> “An *autonomous agent* is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.”
> *Stan Franklin and Art Graesser* [FG96]

This definition requires an agent to have at least the following properties:

- Autonomous: they exercise control over their own actions.
- Reactive: they respond in a timely fashion to changes in the environment.
- Goal-oriented: they do not simply act in response to the environment.
- Temporally continuous: they are continuously running processes.

The agents we are using in this thesis are software agents with all these properties and additionally they are communicative, meaning that they are communicating with other agents to be able to achieve a common goal. We will go into more detail when we introduce the different parts of the architecture of APROCO.

2.1.1 Agents versus Active Objects

There is a blur between the notions of agents and active objects that has been the source of lively discussions. For some, agents are active objects with some intelligent behavior, while others see the difference mainly in active objects being the technology to implement agents as the modeling abstractions. In fact, we are using active Java objects (an object encapsulating an active thread of control) for the implementation of the coordination medium APROCO.

2.1.2 Agent Actions and Configurations

The agents we use in APROCO are interacting to achieve a common goal. We state some definitions taken from [Kie97] that we use throughout this thesis.

Action. The actions performed by agents can be divided into two different classes:

1. *Inter-agent actions:* theses actions perform the communication between different agents. They are the subject of coordination models.
2. *Intra-agent actions:* these actions are all actions belonging to a single agent. They contain computations as well as all communication of an agent outside the coordination model, such as primitive I/O operations or interactions with users.

Configuration. A *configuration* consists of

- a set of interacting agents,
- a set of data spaces, and
- the connections between these agents and these data spaces.

2.2 Coordination

We all have an intuitive understanding of the term “coordination” as something that has to be paid attention to when active entities (mostly people) interact to achieve some result. When we watch a winning soccer team, we may notice how well coordinated the actions of a group of people can be. However, coordination is most clearly visible when it is lacking, and as a consequence the trainer of the soccer team in our example gets fired.

Coordination as a research topic has got a lot of attention and is being investigated in a large variety of contexts, such as computer science, organization theory, management science, psychology, and economics. Malone and Crowston [MC94] made an overview of the different approaches in this different scientific fields in their interdisciplinary study of coordination.

Some frequently used definitions of coordination:

> "Coordination is the additional information processing performed when multiple, connected actors pursue goals that a single actor pursuing the same goals would not perform."
> *Thomas W. Malone* [Mal88]

> "Coordination means managing the inter-agent activities of agents collected in a configuration."
> *Thilo Kielmann* [Kie97]

> "Coordination is the integration and harmonious adjustment of individual work efforts towards the accomplishment of a larger goal."
> *B. Singh* [Sin92]

> "Coordination is the act of managing dependencies between activities."
> *Thomas W. Malone and Kevin Crowston* [MC94]

> "Coordination is the process of building programs by gluing together active pieces."
> *David Gelernter and Nicolas Carriero* [GC92]

These definitions show that coordination is concerned with

- managing the communication which is necessary due to the distributed nature of the application and
- the composition of concurrent systems.

Because coordination as research area is quite new, no convergence of definitions can be found yet. It is still unclear which topics clearly belong to this research area and which clearly not. However, in this thesis we focus on the first two definitions given by Thomas Malone and Thilo Kielmann respectively.

2.2.1 Coordination Models and Languages

Coordination of agents can be expressed in terms of coordination models and languages. We clarify these two different notions in the following.

Coordination Models

A *coordination model* defines how active pieces (i.e. agents) interact and how their interactions can be controlled. This covers the aspects of creation and destruction of agents, communication among agents, spatial distribution of agents, as well as synchronization and distribution of actions over time.

A coordination model consists of three parts [Cia96]:

- **Coordinated Entities (Components):** These are the building blocks which are coordinated. Examples: agents, processes, active objects, tuples, atoms.
- **Coordination Media (Connectors):** These are the media enabling communication between the agents. Coordination media can serve to aggregate a set of agents to form a configuration. Examples: channels, shared variables, data spaces, bags
- **Coordination Laws:** These laws describe how agents are coordinated by making use of the given coordination medium. Example: rule set based on chemical reactions used with Gamma [BLM93].

Coordination Languages

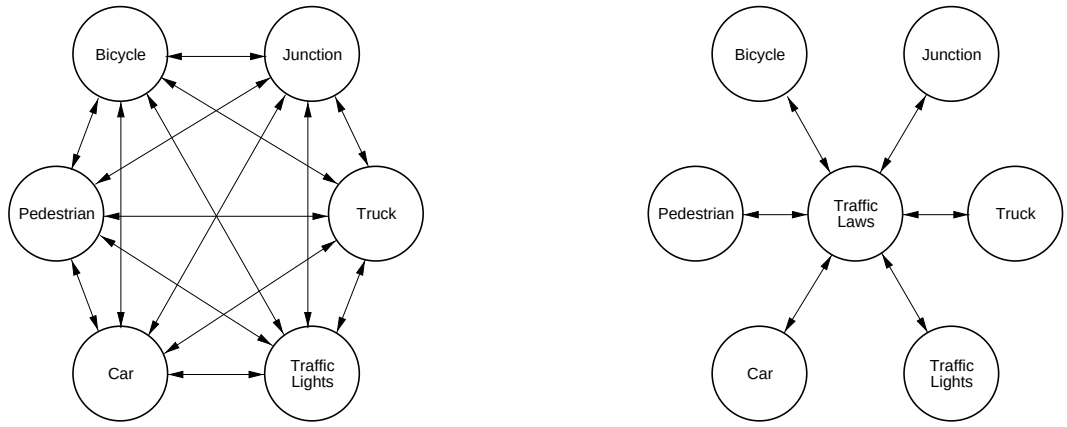
A *coordination language* is the "linguistic embodiment of a coordination model" [GC92], providing a syntactical framework in which a coordination model can be used for realizing applications. We introduce a prominent example of a coordination model, namely Linda, in section 2.2.3, and show an example code using the coordination language C-Linda, which is one of several linguistic embodiments of the Linda model in a computational programming language (C in the case of C-Linda). Hence, a coordination model provides the semantics whereas a corresponding coordination language provides a syntax to use the model within an implementation.

2.2.2 The Problems with Coordination

Coordination of active entities has often been studied in connection with object-oriented software development [AB92, AF+94, FA93]. Several authors have pointed out the lack of constructs in conventional object-oriented programming languages for coordinated object behavior. The message send model that the object-oriented paradigm is based on, is considered too low-level to express coordinated behavior of multiple objects [AB92]. We take the example from [AB92] to show the main problems with multi-object coordination.

Figure 2.1(a) shows the interaction pattern of a possible specification of a system describing a traffic junction using a current object-oriented method. This illustrates a number of problems:

- **No abstraction.** The coordination solution used to coordinate the different objects are "hard-wired" into the participants. In the example in figure 2.1(a) the traffic laws that we use for crossing a traffic junction have to be coded into all the different participants.
- **No extensibility.** The coordination solution is spread across the whole application and thus not easily extensible. In figure 2.1(a) the traffic laws cannot be easily extended if say a new kind of participant is added that needs to be treated in a special way.
- **No flexibility.** The coordination solution cannot be easily changed due to the absence of an explicit representation. In figure 2.1(a) the traffic laws cannot be easily changed because they are coded into all the different participants.
- **No reuse.** It is not possible to reuse the implemented coordination solution separated from the participating objects. If we want to reuse the traffic laws used for the system shown in figure 2.1(a) we cannot do it separated from the participants.



(a) Distributed to the objects

(b) Abstracted within a module

Figure 2.1: Coordinated behavior in the object-oriented paradigm: two possible solutions for the design of a traffic junction

- **No easy enforcement of laws.** Even if all the participants hold a full set of the actually valid traffic laws, no one can play the role of an independent "police" entity to enforce those laws. Every participant is only able to enforce its own behavior to adhere to the laws, but this is not a real enforcement. In our example in figure 2.1(a) the enforcement of the traffic laws cannot be easily done, because they are spread among the participants.

This list of problems reveals the need of an explicit representation for the coordination between active entities as shown in figure 2.1(b). It also shows the difficulties that arises from such an undertaking.

- The coordination code is inside the protocol of the participant entities.
- The coordination code is usually intermixed with application code.
- The coordination code is spread all over the whole application, because it affects more than one single entity.

To be able overcome these problems and limitations we define an architecture which allows us to express coordination abstractions in explicit entities that can be reused and adapted. We construct this architecture and the coordination medium APROCO that we describe in the next chapter using shared data spaces and the basic Linda operations applied on forms instead of tuples. Thus we present a short introduction in Linda and show its limits.

2.2.3 Linda

Linda [CG90] is a coordination model based on processes connected to a shared data space. As mentioned above, a coordination language is the syntactical embodiment of a coordination model.

The coordination language of the coordination model Linda is a set of primitives used within a usual programming language. Because coordination is thought of as an orthogonal aspect to computation, a coordination language is only used to express the coordination part of an application whereas the "host" programming language is concerned with the computational part inside the agents constituting the application. Because Linda consists only of a small set of simple coordination primitives that are independent of the host programming language, it has been added to a large variety of different programming languages, such as C, Pascal, Ada, Prolog, Lisp, Eiffel and Java to name but a few (see e.g. [PA98] and [Kie97] for a survey). Linda was originally developed for parallel programming but proved to be a much general approach for coordinating active entities.

Linda has often been referred to as "high-level coordination model" (e.g. in [BCG97]) but this is only true if compared to low-level communication models that are seen as coordination models such as the Message Passing Interface [SO+96] used to program parallel systems which is the original home domain of Linda. What makes Linda appealing is not primarily its level of abstraction, but its simplicity.

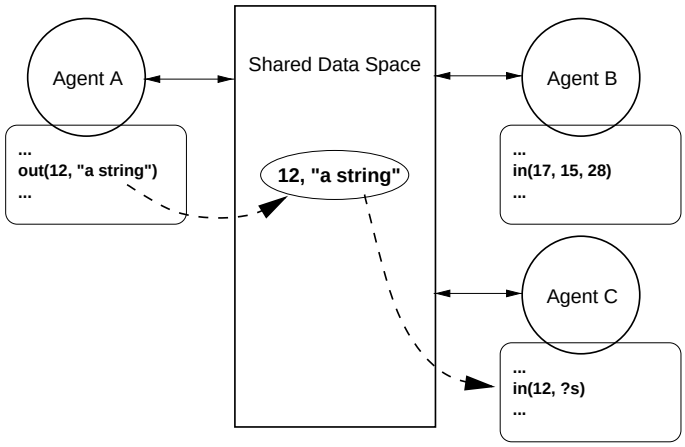


Figure 2.2: Linda in action: three agents communicating through a shared data space

Linda is able to express simple coordination - via producing and consuming data items (tuples) in a shared data space (called *tuple space*) as shown in figure 2.2. Linda offers basically four operations for this purpose:

- `out(t)` inserts the tuple *t* in the data space with no regard whether such a tuple already exists or not. The executing process continues immediately.
- `in(p)` causes some tuple *t* that matches the *pattern tuple p* to be withdrawn from the data space. A pattern tuple[1] is a series of typed fields; some are values, others are typed place-holders, indicated through a prefixed question mark. Once `in(p)` has found a matching *t*, the values in the tuple *t* are assigned to the corresponding place-holders in the pattern tuple *p*, and the executing process continues. If no matching *t* is available, the executing process

[1]The original name for the pattern in [CG90] is *anti-tuple*, but since an "anti-thing" is usually not matching in any sense with the "thing", we chose a different name for it.

suspends until one is, then proceeds as before. If many matching tuples are available, one is chosen arbitrarily.

- `rd(p)` has the same behavior as `in(p)` except that the matched tuple *t* remains in the data space.
- `eval(t)` is the same as `out(t)`, except that *t* is evaluated after it enters the data space: `eval(t)` implicitly creates one new process to evaluate each field of *t*. When all fields are completely evaluated, *t* becomes an ordinary passive tuple.

There are two more primitives that are variants of the `in` and `rd` operations with non-blocking (and non-clear) semantics:

- `inp(p)` attempts to locate a matching tuple and returns `false` if it fails; otherwise it returns `true`, and performs the assignment of values to place-holders as with the blocking `in` operation. It is not clear from the description in [CG90] whether a matching tuple is removed from the tuple space or not.
- `rdp(p)` has the same behavior as `inp(p)`. Here again, it is not clear, whether a matching tuple is removed from the tuple space or not.

An Example: Dining Philosophers

The famous example of the "dining philosophers" originally posed by Dijkstra is a classical problem in the domain of concurrent and parallel programming and can be found in many textbooks. Thus, we just present it the short way[2]: a round table is set with some number of plates (usually five); there is a single chopstick between each two plates, and a bowl of rice in the center of the table. Philosophers think, then enter the room, eat, leave the room and repeat the cycle. A philosopher can only eat with the two chopsticks to the left and to the right of the plate he is seated in hand. If all philosophers simultaneously grab, say their left chopstick, no right chopsticks are available and no philosopher can eat, so deadlock occurs.

We present a deadlock-free solution for this classical textbook example using Linda taken from [CG90]. As mentioned above, Linda is a genuine coordination model and can only describe the coordination aspects of an application. For an application such as this example we also need to describe the computational aspects and for this purpose we need a normal programming language. In this example the programming language C is used for the computational tasks. There are `Num` philosophers in total. The solution uses a "room ticket" to only let `Num - 1` of them into the dining room at the same time to prevent the mentioned deadlock situation.

The `eval` operation in the code shown in figure 2.3 is not really used as it should be according to the just presented introduction. Usually, the `eval` operation ends up in a passive tuple in the shared data space that can be consumed with an `in` operation later on. This is not the case in the shown example: the philosopher processes called `phil(i)` run forever, thus they cannot

[2]This version of the dining philosophers is derived from the one posed by Dijkstra in which a philosopher was thought to be unable to eat spaghetti just with one fork, he needed two forks to do this. Because this was not very realistic even for philosophers, Professor Ringwood changed the example to philosophers eating rice with two chopsticks. Of course, this did not really answer all the questions.

```
phil(i)
  int i;
{
  while(1) {
    think();
    in("room ticket");
    in("chopstick", i);
    in("chopstick", (i+1)%Num);
    eat();
    out("chopstick", i);
    out("chopstick", (i+1)%Num);
    out("room ticket");
  }
}

initialize()
{
  int i;
  for (i = 0; i < Num; i++) {
    out("chopstick", i);
    eval(phil(i)); See remark in the text
    if (i < (Num - 1) out("room ticket");
  }
}
```

Figure 2.3: Dining philosophers in Linda (using C as computation language)

eventually end up in a passive tuple. Nevertheless, the `eval` operation has to be used in our example, because it offers the only way in Linda to create concurrent processes. It it therefore not a “hack” here, it is the standard idiom to be used, even if there is no resulting passive tuple at the end of the processing of the operation.

Note that the solution presented in figure 2.3 is not fair, in the weak sense of fairness, meaning that if a process continuously makes a request, eventually it will be granted: it is possible that a slow philosopher remains blocked on an `in("room ticket")` statement while a speedy one repeatedly `out`s a room ticket and then grabs it again, leaving the slow philosopher still blocked. We can of course solve the problem - as done in [CG90] - by stating that the the implementation of the operations has to ensure fairness of this kind. If we know that the programming language cannot guarantee this (like e.g. in Java[3]) we have to change the solution accordingly.

[3]The scheduling of threads is not guaranteed to be fair in any Java run-time system. Threads with the same priority are not necessarily preempted in favor of each other. See [Lea97] for details.

Properties

Linda embodies three main principles with its primitive operations:

- **Anonymous communication.** The producer of a data item does not need to know the consumer and vice versa.
- **Universal associative addressing.** Data space items - tuples - are addressed by the values of some of their fields and not by their location in the tuple space.
- **Independent data.** The data stored in the tuple space is independent from its creator process.

This properties significantly contribute to the suitability of Linda for the use in dynamically changing environments, i.e. open systems. The simplicity of its operations allows an easy mapping of low-level coordination problems like synchronization of concurrent processes.

The simplicity is what makes Linda a perfect basis for a coordination architecture that offers higher-level coordination abstractions to the user. This thesis presents such an architecture.

Limits

As can be seen in the presented example in figure 2.3 it is easy to describe a particular solution to a coordination problem in Linda, especially if it is a quite low-level coordination problem such as synchronization. Nevertheless, Linda has some serious problems and defects that needs to be addressed in a system that is meant to allow one to easily express flexible and reusable coordination abstractions.

- Linda is not a concrete language. Linda is a set of simple “add on” primitives that are easy to understand and use, and fit into almost any computational model. This in contrast means that Linda offers only a very limited functionality. One still has to program realistic coordination abstractions on top of the ones offered by the model.
- The coordination code is not strictly separated from the computational code within a component. The conceptually different parts of the functionality of a component are intermixed and thus not easily adaptable and reusable independent from each other.
- The coordination solution is spread all over the participating components. Linda does not enforce the designer of an application to strictly separate the coordination components from the computational components. Although the coordination is explicit through the use of coordination primitives offered by Linda it is still the most common design to use coordination intermixed with computation in the same component. This prevents one from successfully reusing the coordination solution in different settings. Furthermore, an adaption of the coordination behavior to changed requirements is not feasible, because it requires the adaption of every single component of the application. The Linda example shown in figure 2.3 is not necessarily fair as mentioned above. If fairness is a requirement for the application, we have to change every participant’s code to realize it. In this special case we are lucky, because all the participants share the same code and thus we only have to change it in a single location,

but in a more general situation, we would have to change every participant's code to adapt the used coordination solution.

These limits pose serious problems to the construction of flexible and reusable coordination solutions in the domain of open systems. This directly results in the need to express the coordination abstractions in explicit entities. These entities can still use Linda's shared data spaces and its primitive operations to realize their coordination purposes, while being explicit they offer flexibility and reusability. This combination yields the advantages of the decoupling of participants offered by Linda and the flexibility and reusability offered by explicit entities. Furthermore, using explicit entities allows the coordination code needed in the participants to be reduced to a minimum. This results in more reusable participant components as well.

2.3 Related Work

We compare the main ideas of APROCO with the most prominent coordination models and with the work that directly influenced APROCO.

2.3.1 Linda

We presented a detailed description of Linda [CG90] in the preceding section 2.2.3. Linda is based on processes (agents) connected together via a single shared data space. The communication and synchronization between these processes is achieved using a small set of primitives that allow a process to generate a tuple to be stored in the shared data space, inspect the shared data space for a specific "type" of tuple, or consume such a tuple from the shared data space. These operations can either be done in a blocking or a non-blocking manner. The creation of new processes can be done by generating a special type of tuple containing expressions to be evaluated concurrently. Linda offers separation of coordination aspects from computational aspect of a process on the level of single statements in its program code.

APROCO uses Linda's decoupling of agents using shared data spaces and its primitive operations for generative communication. APROCO extends Linda by offering the means to strictly separate coordination entities from computational entities by introducing coordination agents (see section 3.4.3). This allow us to adapt or reuse the coordination entities independently from the computational entities, which is a prerequisite for the needed flexibility and reusability. Using coordination agents as implementations of coordination abstractions APROCO offers the means to construct useful coordination solutions on top of Linda's primitive operations with a higher level of abstraction from the concrete situation. APROCO extends Linda's notion of tuples by incorporating forms as the basic carriers of information. This results in easy extensibility of the applications as we will show in section 3.8.6.

We want to point out that Linda by itself is sufficient to solve most of the problems in the domain of coordination. The model is simple and easy to understand and offers important properties such as decoupling of agents and associative addressing. However, Linda does not enforce the user to strictly separate coordination and computation on the level of separate entities. This is a major obstacle for flexibility as well as reusability of such entities. In this sense APROCO extends Linda with a natural way to cleanly separate coordination from computation using coordination

agents and furthermore makes life easier for the user by offering extensible forms instead of the rigid tuples.

2.3.2 Objective Linda

As mentioned in section 2.2.3 Linda can be embedded in a variety of programming languages, including object-oriented ones. Objective Linda [Kie97] is a coordination model that seamlessly incorporates object-oriented concepts into the Linda model. Objective Linda is built for object-oriented parallel programming.

APROCO incorporates a lot of ideas from Objective Linda. The dynamic composition model of agents dynamically connecting to and detaching from multiple shared data spaces is adopted from Objective Linda and from the related approach of Tom Holvoet [Hol97]. APROCO shares the notion of agents with Objective Linda. Agents are the active entities whose interactions need to be coordinated by the model. Objective Linda is carefully defined as being a coordination model suited for coordination solutions for open systems while APROCO offers an architecture for explicit, flexible and reusable coordination abstractions as coordination agents on top of such a model. This means that APROCO can be seen as a natural extension to Objective Linda rather than a competitor.

2.3.3 Programmable Coordination Media

APROCO is strongly influenced by a paper from Denti, Natali, and Omicini [DNO97] in which they present their idea of *programmable coordination media*. In their work, the shared data space used for the global communication can be programmed to react differently on different communication events. The connected agents are generating communication events that are trapped by the programmable coordination medium and handled according to its program. To be able to do this, the reactive tuple space was designed as full-fledged logic theory. The reactive behavior of the tuple space can be changed by changing its program using "reaction programming". Reaction programming is based on the notion of reaction (borrowed from the chemical metaphor used in Gamma [BLM93]), triggered in response to logical events' occurrence, and specified through special tuples of the form `react(Event, Body)` in the shared data space. The reaction body `Body` is the collection of the primitive operations to be executed when the logical event `Event` occurs. Reactions allow the medium to perform operations on the tuple space items in a way that is not perceptible for the connected agents. They can only see the final effect of the whole reaction. Thus, reaction programming is a straight-forward way to implement transactional behavior on a shared data space.

The behavior of the coordination medium APROCO can be changed by adding, adapting, or exchanging its coordination agents. In contrast to the reactions used by Denti et al. the coordination agents in APROCO are pro-active, i.e. they can initiate operations on shared data spaces on their own behalf. This enables the coordination medium to offer services to its clients that go beyond the capabilities of communication events such as time-dependent behavior. APROCO also hides intermediate states of shared data spaces from its clients because the coordination agents use internal shared data spaces that are not accessible to the client agent. Furthermore, APROCO can fulfill its coordination purposes in a much easier and cleaner way by only using one communica-

tion paradigm, namely generative communication, and one abstraction for activity, namely agents compared to the different concepts needed by the programmable coordination media from Denti et al.

2.3.4 Coordination Components

APROCO is also influenced by the work of Sander Tichelaar [Tic97]. He describes a component-based approach to coordination as opposed to the language-based approaches such as Linda. His work shows the need for separating the coordination from the rest of the application on a component level. This separation is adopted by APROCO by using coordination agents as explicit representation of the coordination aspects of the application. Coordination Components are based on message passing in object-oriented systems. APROCO is more general and uses generative communication between agents that can be implemented in any programming language. The coordination entities used in APROCO are "enhanced components", they are agents and thus can have pro-active behavior.

The concept of exchangeable (pluggable) coordination policies is taken from the work of Sander Tichelaar. In his solution the policy can be supplied as a parameter at application setup. In APROCO the policy used for a specific action from a coordination agent can be exchanged dynamically at run-time.

2.3.5 Synchronizers

Synchronizers [FA93] are special objects able to express coordination patterns within a multi-object language framework based on specifying and enforcing constraints that restrict invocation of a set of objects. They support the separation of concerns and the reuse of coordination code. The separation of coordination code into a special object is similar to the coordination agents in APROCO. Synchronizers are abstracted from the underlying programming language and the protocol used to enforce the required object properties. The coordination agents in APROCO are independent of the programming language as well, but they are based on generative communication as the means to control interactions, opposed to the message-passing communication that synchronizers are based upon. Synchronizers cannot be changed dynamically and their connections to the participating objects are static. In APROCO the coordination agents can be adapted dynamically and they have no direct connections to the participating agents, thus they are operating on a higher level of abstraction. Furthermore, coordination agents are not bound to constraining the coordination participants, they can also pro-actively initiate some actions.

Chapter 3

Approach

In this chapter we are presenting the architecture of our coordination medium APROCO and explain and justify some design decisions we made. The first section lists the requirements for a coordination medium suited for the domain of open systems.

Section 3.2 describes the architecture of APROCO consisting of client agents and the coordination medium itself. The client agents communicate using generative communication. The coordination medium is itself built out of coordination agents that communicate with each other over shared data spaces and coordinate the activities of the client agents by inspecting or transforming the flow of data items between the client agents. All data is wrapped into forms to allow the system to be easily extended.

Section 3.7 shows a first simple example of a coordination problem solved within APROCO. We show the different parts of the architecture and how they work together.

In section 3.8 we explain and justify design decisions we made. We present the dynamic composition model incorporated into APROCO to allow dynamic composition and re-composition of an agent configuration. We explain why we used multiple shared data spaces to implement APROCO instead of just a global one for all the agents. We discuss the advantages of designs using private data spaces, and access rights on data spaces in general. The section is concluded by a presentation of forms and an explanation why they provide extensibility.

Section 3.9 summarizes the main properties of APROCO and section 3.10 concludes the chapter with an evaluation of our approach.

3.1 Requirements

From the discussion of open systems and coordination in the preceding chapters a list of requirements result that a coordination medium suited for this domain must fulfill.

- Clean separation of concerns
- Explicit representation of the coordination solution
- Reusability of this representation
- Flexibility

- Decoupling of the active entities
- Dynamic re-composition of the active entities

In the following sections we present the coordination medium APROCO that fulfills these requirements.

3.2 Architecture

The *coordination medium* APROCO is built upon *shared data spaces* as used in the coordination language Linda [CG90]. In short, shared data spaces allow for a special communication style - called *generative communication style* [Gel85] (see section 2.2.3 for an introduction into Linda) - to be used for the connected agents to communicate with each other. With generative communication the sender generates data items and stores them in a data storage accessible to both the sender and all possible receivers. The receiver can then inspect this data storage for specific data items and read or consume them. This shared data storage is mostly called shared data space, but sometimes also *blackboard* [BMR+96].

The data items stored in the shared data spaces are *forms* in our case, opposed to the normally used tuples. A form as we use it here is a set of bindings of labels with corresponding values. Forms allow one to build easily extensible systems, as we discuss later in this chapter.

The customers of the coordination medium are called *client agents*. They are active entities that have the necessary properties to be called autonomous agents according to the definition in [FG96] presented in chapter 2. A client agent is connected to the coordination medium through at least one shared data space. All data items - forms in our case - inside a shared data space are potentially accessible to a client agent that has got a reference for this data space. An agent can get a data space reference either statically at compile-time or dynamically at run-time. Such a data space reference can be wrapped into a form and exchanged through another data space.

The coordination between the client agents of the coordination medium is performed by the *coordination agents* inside the medium. A coordination agent has the same properties as a client agent and operates on data spaces the same way as a client agent. The only difference lies in the privileged location of the coordination agent, it resides “inside” the coordination medium and thus has more information about the configuration of the whole application. A coordination agent encapsulates the solution for a specific coordination problem, abstracted from the concrete participants. A coordination agent use generative communication to read out, transform existing forms or add new forms to to the flow of forms between the client agents, thus realizing its coordination purpose. APROCO can have an arbitrary number of coordination agents that can be combined to provide the client agents with the “coordination services” they need, e.g. security or fault-tolerance. Coordination agents can be combined by connecting them together using multiple data spaces arranged according to the needs of the application.

A collection of interacting agents (both client and coordination agents) together with their connections is called a *configuration*. The actual configuration can be changed dynamically in APROCO, i.e. client agents can enter and leave a configuration at any time.

APROCO is a *programmable coordination medium* [DNO97], because it is possible to add, change or remove coordination agents that are part of the medium and together constitute its

coordination services for the clients. If we change the coordination agents we change the behavior of the coordination medium as a whole and its reactions to the client agent's communication. This means that the medium as a whole is programmable. In fact, the coordination medium APROCO is pro-active and thus more powerful than the reaction programming used by Denti et al. [DNO97].

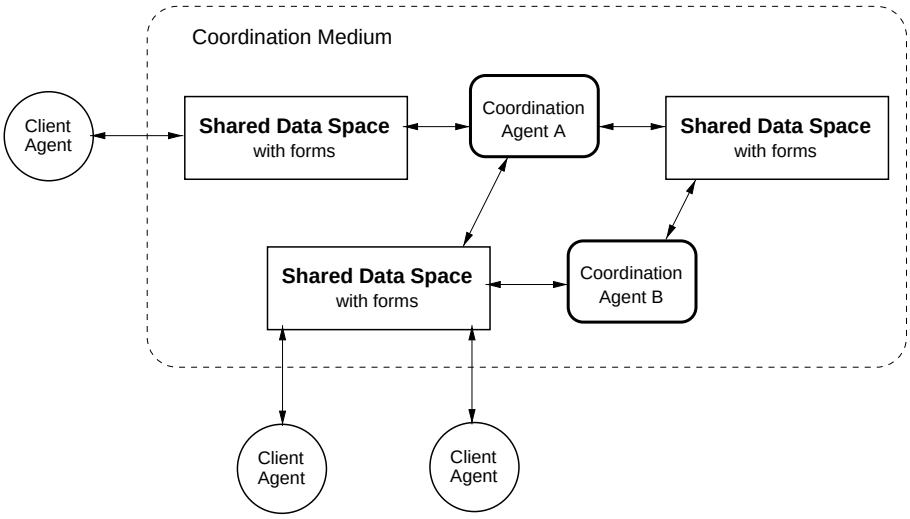


Figure 3.1: Overview of the coordination medium APROCO

The general architecture of APROCO is shown in figure 3.1. The arrows denote the allowed communication paths using generative communication. We will introduce the different parts of the architecture and their relationship in the following subsections.

3.3 Clients

The clients of the coordination medium APROCO are active, self-contained entities performing actions on their own behalf. According to the definition and classification found in [FG96], these entities are *autonomous agents* with the following properties[1]:

- Autonomous: they exercise control over their own actions.
- Reactive: they respond in a timely fashion to changes in the environment.
- Pro-active: they do not simply act in response to the environment.
- Temporally continuous: they are continuously running processes.
- Communicative: they communicate with other agents.

[1]Agents often have connotations of being intelligent or mobile, but we are not interested in these aspects in this thesis.

All clients are connected to the coordination medium. In fact, each client has at least one reference to a shared data space and uses Linda-like communication primitives [CG90] to communicate with the medium and thus with the other clients. All communication between clients takes place via the coordination medium using the following Linda-derived communication primitives[2]:

- `out` inserts a form in the data space with no regard whether such a form already exists or not. The executing agent continues immediately.
- `in` removes a form that "matches" the provided pattern form from the data space and reads in the values of this form. If such a form could be found in the data space the executing agent continues immediately. If no matching form is available, the executing agent suspends (in the blocking version of this operation, see next paragraph). As soon as a matching form gets available, the suspended agent is woken up and continues. If more than one matching form is in the data space, one is chosen arbitrarily. For details on matching operations see section 3.8.6.
- `read`[3] has the same behavior as `in`, except that it is non-destructive.

The `in` and `read` primitives can both be called in a blocking or a non-blocking manner. The blocking operations block the calling agent, waiting for a matching item to be inserted into the data space by another agent with an `out` operation. The non-blocking operations return immediately with an indication whether the operation was successful or not (e.g. a boolean return value). The unclear semantics of the non-blocking `inp` and `rdp` operations in standard Linda as described in section 2.2.3 is remedied by the operations we chose in APROCO:

- `in_nb` attempts to find a matching form in the data space and removes this form from the data space and returns it as the result of the operation if successful; otherwise it returns `null`. The executing agent continues immediately in both cases.
- `read_nb` has the same behavior as `in_nb`, except that in the case of a successful matching of a form, it remains in the data space.

Tasks

A client of APROCO can perform the following tasks that are relevant to the coordination medium. A client can:

- **produce** (with `out`), **read** (with `read`), and **consume** (with `in`) forms from the data spaces it has got references for - either statically through the programming language or dynamically acquired as shown later - and can this way participate in the overall tasks of the application.

[2]The names for the operations are taken from original Linda [CG90]. Note that they take the agent's perspective: to put an item into a data space, an agent has to call the `out` operation, and to take an item out of a data space, it has to call the `in` operation.

[3]The original name in Linda for this operation was `rd` but for the sake of "readability" we chose to change it into `read`.

- **create** new data spaces on its own and perform the same operations on them as on those data spaces that it initially had the references for.

- **expose** its own created data spaces for other clients to attach to them. This is done by wrapping a reference to this data space into a form and putting it with an `out` operation into a well-known data space representing a global environment, the *global configuration data space*.

- **attach** itself to existing data spaces. This is done with an `in` or `read` operation on the global configuration data space to get a form with the reference for the desired data space. If the data space was created by another client, it has first to be exposed before the client can attach to it.

Besides those tasks a client can perform whatever tasks it likes, as long as they do not interfere with the medium, as would e.g. direct method invocations or a different kind of interaction with the medium or other clients, than listed above.

Life cycle

Client agents are created in APROCO by instantiating and activating an object possessing its own thread of control[4]. To be a useful part of the whole application, an agent needs to get access to at least one already existent data space. This can either be done statically at compile-time or dynamically at run-time by attaching to an exposed data space.

```
/* Create a voter agent */
new Thread(new Voter(globalspace)).start();
```

Figure 3.2: Java code for the creation of a typical client agent

The creation of a typical client agent in APROCO is shown in figure 3.2. The parameter of the example client agent is a standard reference for the global configuration data space that is - among other things - needed for the dynamic acquisition of data space references.

A client agent is active as long as it has not deliberately stopped its processing or its execution environment has terminated the execution. It can be either running and performing tasks as mentioned before or be blocked because of an `in` or `read` operation that could not yet deliver a matching form from a data space.

Client agents cannot be terminated by other agents. They either terminate themselves or are terminated by the execution environment, in our case the Java virtual machine. It is up to the agent to perform some cleaning up tasks before termination.

[4] In standard Linda the `eval` primitive is used to dynamically create new processes. This operation creates *active data items* in the data space that are processed and eventually end up as normal passive data items. We simulate this behavior by creating a new Java thread that eventually generates a form containing the result of the processing.

3.4 Coordination Medium

The medium as shown in figure 3.1 consists of shared data spaces and coordination agents that operate on them to deliver their "coordination services" to the client agents. The clients are directly connected to some of the shared data spaces inside the medium, so the border line of the medium is really only a conceptual one.

3.4.1 Shared Data Spaces

The building blocks of the coordination medium are the shared data spaces used for communication and coordination purposes. A shared data space is a multi-set of items, i.e. identical items may exist in a data space. As mentioned we use forms as data space items in APROCO. Agents can only communicate with other agents through at least one shared data space. They use generative communication to interact with each other to achieve the overall goal of the whole application. The shared data spaces we employ in APROCO are derived from the coordination language Linda [CG90] and use its primitives to generate, read, and consume forms. All these operations are atomic and work on a single form. The used primitives are as mentioned earlier:

- `out(f)` inserts form *f* into the data space. The executing agent continues immediately.
- `in(p)` removes a form *f* that matches the provided pattern form *p* from the data space, and reads in the values from *f*. If such a form could be found in the data space the executing agent continues immediately. If no matching form is available, the executing agent suspends (in the blocking version of the operation). As soon as a matching form gets available, the suspended agent is woken up and continues. If more than one matching form is in the data space, one is chosen arbitrarily.
- `read(p)` has the same behavior as `in(p)`, except that the matched form remains in the data space.

The `in(p)` and `read(p)` operations can also be used in a non-blocking variant called `in_nb(p)` and `read_nb(p)` as described in section 3.3.

Forms and pattern forms. A *form* is a set of bindings of labels (or keys) with corresponding values. The values can be of arbitrary type, even another form is a possible value. The `in` and `read` operation requires a pattern form to be supplied as parameter. A *pattern form* is a normal, possibly empty form. This pattern form is used to find a matching form in a data space using pattern matching with a matching operation defined for forms. The matching operation we used in APROCO is very simple: a pattern form *p* matches a form *f*, if the labels of *p* are a subset of those of *f*, and the corresponding values of these labels are identical.

A detailed discussion of forms and the matching operation can be found in section 3.8.6. An short introduction and an example of a form including the program code is presented in section 3.4.2.

Main properties of shared data spaces. The main properties of shared data spaces are:

- Anonymous communication: an agent only puts items into a shared data space or consumes data items from there.
- Associative addressing of data: the data is read or consumed through a pattern that is provided and not e.g. through its "position" in the data space.
- Independent data: the data lives independently from its creator.

Constraints imposed by the used Linda implementation.

- All operations work on a single form. It is not possible to `in` or `read` more than one form atomically.
- If more than one form matches a pattern form *p*, one is chosen arbitrarily. This means, that subsequent `read` operations can return the same form, even if more than one matching form would exist in the data space. This problem is known as the "Linda multiple `rd` problem" [RW96]. Thus, it is not possible to read all the matching forms in a data space simply by repeatedly calling the `read` operation until the same form will be returned twice.
- There is no support for security in standard Linda. Linda does not offer means for access rights on tuples in the data space. As we will show later, it is necessary to have access rights to be able to establish security.

Private data spaces. A client agent can create new data spaces on its own. These data spaces are initially "private", i.e. no other agent has access to them except the one that created them. To play any useful role in the whole application, a data space needs to be accessible at least to the coordination agents inside the coordination medium. This can be done by exposing the data space to other agents as described before. If a data space is only used by one client agent and the coordination medium but not by other client agents, we call it a *private data space*. Private data spaces are shared data spaces as well, but they have only one client agent attached to them (see the multicast example in section 3.7 for a first example of private data spaces).. It is up to the application programmer to decide whether a specific data space should be a shared or a private data space depending on the application design. As soon as another client agent has attached to a private data space we cannot call it private any more. We discuss the advantages of designs using private data spaces in section 3.8.4.

Coordination using a shared data space. Shared data spaces and the concept of generative communication can be used to coordinate active entities. To illustrate this, we show a simple example of a shared resource used to solve the problem of mutual exclusion of critical parts of two processes.

Figure 3.3 shows two processes that synchronize over a shared data space holding a shared resource, in this case simply a string object. The situation shows process A running, because it could grab the string object from the shared data space, while process B is blocked on the

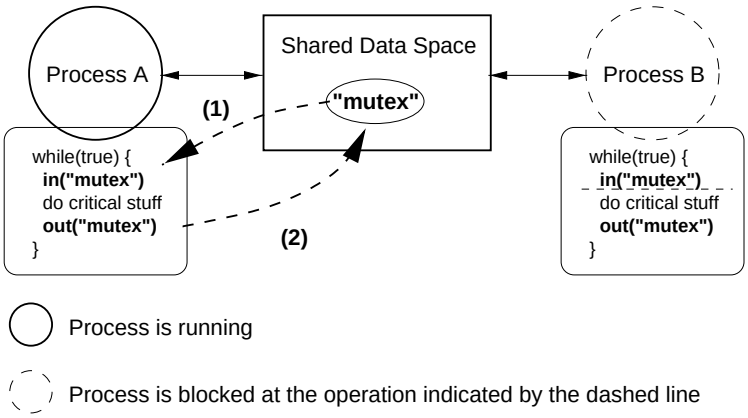


Figure 3.3: Coordination using shared data spaces: mutual exclusion

`in("mutex")` operation waiting for the shared resource to be put back again. A new process that has to protect its critical part against the other ones can simply be connected to the same shared data space and use the same operations for coordination.

Note that the solution shown in figure 3.3 is not fair, in the weak sense of fairness, meaning that if a process continuously makes a request, eventually it will be granted: if two processes are blocked, waiting for the same item to appear in the shared data space, one process is nondeterministically chosen and gets woken up. Thus we cannot be sure, that say process A not always grabs the item before process B gets woken up. Of course, we can delegate the problem to the implementation of the primitives we used to access the shared data space and state that they have to be programmed in a way that fairness can be guaranteed.

Evaluation. The main advantage of the use of shared data spaces is the decoupling of producers and consumers of information. This decoupling means anonymity as well as asynchrony. Anonymity, because the producer and the consumer do not have to know each other in order to have successful communication. Asynchrony, because the producer can proceed with its task without having to wait for the consumer to be ready. Both properties combined yield the generative communication style [Gel85] that enables one participant to communicate with another even if the other one is non-existent at the time. Coordination models based on generative communication are well-suited for the domain of open systems [Hol97].

Shared data spaces are suitable to describe solutions for traditional coordination problems such as synchronization, shared resources, etc. These problems have in common that they are quite low-level and thus they map easily to the low-level Linda primitives used with shared data spaces. If we like to describe higher-level coordination problems that have their origin in real-world applications, e.g. transactions, the mapping is much more complicated and not straightforward anymore. This is the reason why we introduced a coordination medium that allows one to describe higher-level coordination abstractions in the form of coordination agents that themselves are built upon the simple Linda primitives.

Global Configuration Data Space

In an open system client agents can join and leave a configuration at any time. To enable the system to keep track of the participating client agents, it is necessary to introduce at least one globally reachable data space for configuration information. This global configuration data space represents a *global environment* as found in many other systems. Client agents that enter an existent configuration have to register with this global configuration data space by providing a reference to their private data spaces, if existent. They can also provide the medium with other useful information for the coordination agents such as the agent's processing capabilities in a "farm of workers" setting to enable a coordination agent (called workload manager) to distribute processing requests according to a specialist parallelism [CG90] based distribution policy.

Coordination services like the just mentioned processing distribution can be realized in a flexible way by storing an explicit representation of the actually used *policy* in the global configuration data space. The term "policy", according to the American Heritage Dictionary [Mor81], means "a general principle that guides the actions taken by a person or group". With this definition, we take a policy to be the general principle that guides the actions taken by a specific coordination agent in our coordination medium. The explicit representation of the policy as a *policy object* enables the clean separation of the policy from its implementation. The policy object is wrapped into a form and stored in the global configuration data space. Every time a coordination agent has to perform an action according to the actual policy the corresponding policy object is consulted. This allows the dynamic exchange of the currently used policy by simply exchanging the form containing the policy object in the global configuration data space.

3.4.2 Forms

A form as we use it in APROCO is a set of bindings of labels (or keys) with corresponding values. These values can be of arbitrary type. Because we are using the object-oriented programming language Java to implement APROCO all objects that can be described in Java are valid values in a form. This especially allows for the construction of nested forms, because a form itself is a valid form value. Note that the basic datatypes in Java must be wrapped into objects to be valid form values.

The use of forms in APROCO is motivated by the extensibility considerations found in [LAN98], where it is shown that with forms it is easy to extend functionality without breaking the existing behavior. This is done by introducing new labels for the new functionality. We show an example of this mechanism in section 3.8.7.

Form Example and Notation

Figure 3.4 shows the Java code for an example form as we use it in APROCO, and the shorter notation that we use throughout the thesis. The code for the form implementation can be found in section A.5. For the implementation of the forms we used an object container from the Java class library JGL [Obj97]. The `HashMap` class we used from JGL is similar to the standard Java `Hashtable` class, but offers more useful functionality.

```
Form f = new Form();
f.put("Type", "RegisterClient");
f.put("ClientID", new Integer(15));
f.put("Handle", privspace);
```

Type = "RegisterClient"
ClientID = 15
Handle = {Run-time object reference}

Figure 3.4: An example of a form as we use it in APROCO and its notation in the text

3.4.3 Coordination Agents

A coordination agent has the same properties as a client agent and operates on data spaces using generative communication the same way as a client agent. The main difference lies in the privileged location of the coordination agents, they are within the medium's boundaries and thus have access to more information about the whole configuration.

A coordination agent provides the client agents with a specific "coordination service" such as fault tolerance. These coordination services represent implementations of specific coordination abstractions. A coordination abstraction is a solution for a coordination problem abstracted from the concrete participants and situation. These coordination services are encapsulated into the coordination agents, because of the agents' ability to behave *pro-actively* instead of only reactively to the environment. Pro-active behavior is necessary for services that need more information that can be available through the inspection of the information traffic that goes over the shared data spaces. If a coordination service needs to perform an action that is not related with the participants actual state such as time dependent actions, such a service cannot be realized using only reactive behavior. The possible pro-active behavior implies that coordination agents have the ability to initiate an action by inserting an appropriate form into a data space without being requested to do so by a client agent. Thus, coordination services in APROCO are operational without being activated by the client agents, if desired.

Tasks. A coordination agent can perform the following tasks that are relevant for the clients of the coordination medium. It can:

- produce, read, transform, and consume forms from **a client's data space**.
- produce, read, transform, and consume forms from a **data space that it shares with another coordination agent** to coordinate with the other one's service.
- attach itself to existing data spaces. This is done the same way as for the client agents.

Access to client communication. A coordination agent does not have any special means to get involved with the communication between client agents than through inspection of the data spaces that it has access to. This implies that - without leaving the generative communication style - it is not possible to give the coordination agent a higher priority to read or consume a particular form in a data space than a client agent competing for the same form. In some cases the client agent would be able to consume the form before the coordination agent could read it or perform some transformations on it. It is thus necessary either to use multiple shared data spaces or to simulate

them on a single shared data space to allow the coordination agents to fulfill their coordination purpose. This simulation can be done using a special identifier (e.g. a string object) in each data space item to show to which logical partition of the single shared data space the item belongs to.

Life cycle. Coordination agents are created the same way as client agents by instantiating and activating an object possessing an own thread of control. To enable a coordination agent to perform its coordination service it must be given access to some shared data spaces that it should operate on by providing it with references for them.

```
/* Create the security coordination agent */
new Thread(new SecurityAgent(agentspace, globalspace)).start();
```

Figure 3.5: Java code for the creation of a typical coordination agent

The creation of a typical coordination agent in APROCO is shown in figure 3.5. The first parameter of the example coordination agent is a reference for a data space that is only used by the coordination agents of this application. The second parameter is a standard reference for the global configuration data space which has the same function as with the client agents.

A coordination agent is active as long as it has not deliberately stopped its processing or its execution environment has terminated the execution. Like the client agents, coordination agents can be running and performing tasks as mentioned before, or be blocked because of an `in` or `read` operation that could not yet deliver a matching form from a data space.

Coordination agents cannot be terminated by other agents. They either terminate themselves or are terminated by the execution environment, in our case the Java virtual machine. They normally run until the whole application is ended.

3.5 List of Used Coordination Agents in APROCO

We offer a number of coordination agents to be used within APROCO. We introduce them here and shortly describe the coordination service they implement. They are grouped according to the number of data spaces they need to be connected to (or will get the data space references dynamically). Note that this list is by no means complete, because a complete list of coordination agents would imply a closed coordination medium and APROCO is an open coordination medium.

1. Operating on one shared data space:

   - **Registration:** The registration agent listens for new client agents registering their presence with the coordination medium through the global configuration data space and transforms this information into special forms for the coordination agents interested in it. The registration agent maintains a list of the currently registered client agents in the global configuration data space and can also send a notification on the

registration of a new client to coordination agents that are waiting for such a notification addressed to them. This is useful for coordination agents that need to react on the registration of a new client agent. Those coordination agents (e.g. the collector agent) need to register their interest in such notifications with the registration agent providing their name for addressing purposes.

The registration agent is first used in the Observer example in section 5.2.

- **Client Information:** The client information agent listens for new client agents registering their presence with the coordination medium through the global configuration data space. The clients are providing the coordination medium with specific information about themselves such as the processing capabilities of a worker agent. This information is maintained by the client information agent as a special form in the global configuration data space for the coordination agents to consult.

  The client information agent is similar to the registration agent, except that it maintains more information about the client agents, but is not able to send notifications about newly registered clients to interested coordination agents.

  The client information agent is first used in the Administrator / worker example in section 5.4.

- **Policy:** The policy agent can dynamically exchange the policy that a coordination agent is actually using for a specific action. This policy is encapsulated into a policy object and wrapped into a form stored in the global configuration data space. The coordination agent that is using this policy consults this policy form every time before it performs the specific action that this is the policy for. To dynamically exchange the actual policy the policy agent only has to exchange this policy form with one that includes the new policy object. The next action of the coordination agent using this policy will be performed using the new policy object. All valid policy objects need to implement the same interface.

  The policy agent is first used in the Administrator / worker example in section 5.4. In this section the mechanism of dynamic exchange of policies is discussed in detail.

- **Order:** The order agent can be seen as a meta-level coordination agent. It is used for the dynamic re-configuration of coordination of coordination agents. The order agent maintains a list with the relative order of the coordination agents that are interested to get the forms that are passed around in the coordination medium using the same shared data space. When a new coordination agent is registering with the medium to get access to this shared data space, it will be dynamically inserted into this list. Every coordination agent is only allowed to consume those forms that are explicitly addressed to it, and after it is finished with its manipulations it has to consult this order list to find out who is next to get the form, and explicitly address the form to it.

  The order agent is not used in one of the examples in chapter 5, because of the very restrictive protocol it requires the coordination agents to use. We discuss the order agent and the mechanism in section 3.8.2 in detail.

2. Operating on two shared data spaces:

   - **Transport:** The transport agent is a very simple coordination agent. It only transfers forms back and forth from one data space to the other. Because of its simplicity it can be used as skeleton to build more complex coordination services with by implementing additional transformations to the transported forms such as an adding an identification to every form to allow the coordination agents to correctly identify it throughout its flow through the medium.

     The transport agent in its simplest form is not used in one of the examples. A variation using a special protocol presented with the order agent is shown in section 3.8.2.

   - **Fault Tolerance:** The fault tolerance agent is connected to two data spaces and transports requests from one data space into the other and transports answers back in the other direction. The fault tolerance agent maintains a timeout for each request form it has transported and creates copies and resends them if there was no response in time. It puts an error response form into the client agent's data space, if a given number of retries were without response. The fault tolerance agent filters out request repeats originating from the client agent due to an own repetition mechanism to prevent the client agent that processes the requests (usually called server agent) to use system resources unnecessarily.

     The fault tolerance agent is only applicable in request / answer (client / server) settings. The client agent that sends the requests must be ready to accept error answers. The fault tolerance agent is first used in the Fault tolerance service example in section 5.1.

   - **Vote Controller:** The vote controller is used within the Electronic vote example described in section 5.3. This example defines rules that all participants have to follow during the vote. The vote controller is used to enforce these rules. It announces new vote rounds to the participants, checking and counting the incoming votes, and announcing the results to the participants again. The transport of the information to and from the client agents is done by other coordination agents. All this information is accessible for the vote controller in a single shared data space. To be able to decide about the outcome of a particular vote the vote controller needs to know the number of participating voters, this information is available in the global configuration data space (maintained by a registration agent). Together with the authentication agent the vote controller prevents the participating agents from cheating.

     The vote controller agent is first used in the Electronic vote example in section 5.3.

3. Operating on multiple shared or private data spaces:

   - **Multicast:** The multicast agent is connected to one data space that it shares with other coordination agents. Its job is to deliver a copy of the forms in this data space into some client agents private data spaces. The group of client agents that are interested in getting those forms can dynamically change, thus the multicast agent needs the list of actually registered client agents and a reference for their private data spaces to be operational. This information is available in the global configuration data space (maintained by a registration agent). Every time a new form is available in the shared

data space to be distributed the multicast agent updates its internal list of registered client agents.

The multicast agent is first used in a variant in the Multicast example in section 3.7. In this example the multicast agent is updating its internal list of registered agents directly with the registration information provided by the client agent themselves instead of with the information provided by the registration agent.

- **Collector:** The collector agent is the counterpart of the multicast agent. It is also connected to a data space that it shares with other coordination agents, but this data space is used to put the forms into that it collects from the client agents private data spaces. The information about the actual group of registered client agents could also be fetched from the global configuration data space, but because the collector agent does not have a form that it can fetch from the shared data space and use as a trigger to update its list of registered client agents, it needs a different mechanism to do so. It needs to get a notification form in the global configuration data space whenever a new client agent registered with the medium. This notification form is provided by the registration agent, assumed that the collector agent has registered with it beforehand.

  The collector agent in its simplest form is not used in one of the examples in chapter 5. The authentication agent presented in the next paragraph is based on the collector agent and used in the electronic vote example in section 5.3.

- **Authentication:** The authentication agent is an enhanced collector agent. It collects forms from the clients private data spaces and transports them into a specific shared data space that is only accessible for the coordination agents to ensure security. Additional to the transportation of those forms the authentication agent assigns a unique identification to every form originating from the same private data space. This identification enables other coordination agents to check the forms for their origin.

  The authentication agent is used in the Electronic vote example in section 5.3 to ensure a part of the security for the votes given by the voter agents.

- **Workload Manager:** The workload manager is connected to two shared data spaces and transports requests and answers back and forth the same way as the transport agent. The workload manager assigns a destination identification to each of the request forms according to a specific policy that is encapsulated into a policy object. This policy object is wrapped into a form and stored in the global configuration data space. The workload manager consults it each time a new request arrives and needs a destination to be assigned to. The used policy possibly needs additional information about the actually available destination agents that is available in the global configuration data space (maintained by a worker information agent). The actual policy can be dynamically exchanged through the policy agent by exchanging the policy form containing the policy object in the global configuration data space.

  The workload manager is used in the Administrator / worker example in section 5.4.

3.6 Relationship Between the Client Agents and the Medium

Client agents. The coordinated behavior of all client agents together forms the behavior of the whole application. To be able to be a useful part of the application a client agent needs to communicate and coordinate with its fellow client agents. The only allowed way of communicating in APROCO is by generative communication using shared data spaces. The only way for client agents to coordinate their actions to achieve the overall goal of the whole application is through exchanging forms over the coordination medium.

Client agents to the coordination medium. Every client agent is connected to the coordination medium through at least one shared data space. Clients can create new data spaces if they like. To be serviced by the medium they have to expose those data spaces in the global configuration data space to enable the coordination agents to attach to them.

Client agents to coordination agents. Client agents put forms into data spaces to get them delivered to other clients. The transport of this forms to the other client agents in the application is done by the coordination agents inside the medium. Because there is no fixed number of services that a single client agent can require the coordination medium (and with that the coordination agents inside) to perform, there is no fixed relationship between the number of clients and the number of coordination agents constituting the whole application.

Coordination agents. All coordination agents in the coordination medium need to get access to the flow of forms that the clients are passing around to be able to fulfill their job. This means that the coordination agents need at least one shared data space in between them to pass around the forms before they get finally delivered to the target client agent. If a coordination agent needs to be in a special position to transform these forms, e.g. the last one to do any changes before the delivery, the design of the configuration needs to be done accordingly.

3.7 A First Example: Multicast

As a first simple example we show a situation similar to the one used in the Observer pattern [GHJV95] where one client agent needs to notify other client agents about something, e.g. changes to its state. It is convenient for the sender agent if it does not need to know what receiver client agents are interested in getting the notifications. APROCO is suitable for open systems where client agents can enter and leave a configuration at any time and the information about the actual configuration is only available to the coordination agents in the medium, but typically not for the client agents. Thus the only place for the information about the actual set of client agents that are interested in getting the notifications in open systems is within the medium.

The presented solution shown in figure 3.6 consists of a sender agent with a shared data space to put the notifications in, a set of receiver agents each with a private data space where they can fetch the notifications from, and a coordination agent (called multicast agent here) that is waiting for new notifications and multicasts them into the receiver client's private data spaces. The arrows denote the direction of the flow of information.

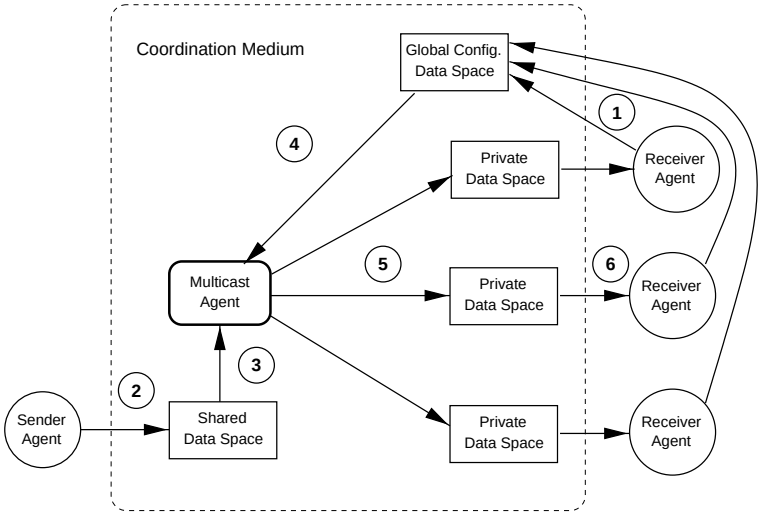


Figure 3.6: Multicast example

Actions. The sequence of actions in this example is as follows:

1. The receiver agents register their presence with the global configuration data space. They supply a reference to their private data spaces that they created themselves. This step is only done once in the application lifetime.
2. The sender agent wraps the information that the notification should carry into a form and puts it into the shared data space.
3. The multicast agent takes the notification form out of the shared data space.
4. The multicast agent inspects the global configuration data space to see if new interested client agents registered in the meantime and updates its list.
5. The multicast agent puts a copy of the notification form into every interested client's private data space.
6. The client agents can fetch the notification form as soon as they are ready to do so.

The easiest solution for the multicast example would have been to use only one shared data space and to connect the sender and the receivers directly to it, as with standard Linda. We discuss the problems that can arise with such a solution in section 3.8.4.

3.8 Precisions and Justifications of Design Choices

3.8.1 Configurations

A configuration is a collection of interacting agents (both client and coordination agents) together with their connections. A useful configuration consists of at least one shared data space, at least

one client agent connected to this data space, and at least one coordination agent connected to the data space. An initial configuration can be set up statically at application startup. The actual configuration can be changed dynamically in APROCO, i.e. client agents can enter and leave a configuration at any time.

Configuration Setup

The initial configuration of an application in APROCO is done by instantiating and activating Java objects and connecting them by passing references for the objects representing the shared data spaces. A typical configuration setup can be seen in figure 3.7.

```
package aproco.examples.multicast;

import jada.*;

public class MulticastExample {

  public static void main(String args[]) {

    ObjectSpace senderspace = new ObjectSpace();    // The sender's data space.
    ObjectSpace globalspace = new ObjectSpace();    // The global configuration data space.

    Form emptyForm       = new Form();              // The empty form matches all forms.
    Form formsToListen[] = {emptyForm};

    /* Create the multicast agent */
    new Thread(new MulticastAgent(globalspace,
                                  senderspace,
                                  formsToListen,    // All forms that the agent listens to.
                                  1)).start();

    /* Create the sender agent */
    new Thread(new Sender(senderspace)).start();

    /* Create the receiver agents */
    Receiver receivers[] = new Receiver[5];

    for (int i=0; i<5; i++) {
      receivers[i] = new Receiver(i, globalspace); // The receivers have an ID just for
      new Thread(receivers[i]).start();            // testing purposes.
    }
  }
}
```

Figure 3.7: The setup of a typical configuration in APROCO

3.8.2 Dynamic Composition and Re-composition

The dynamic change of agent configurations is part of a *dynamic composition model* as described in [Hol97] and necessary as APROCO has to deal with open systems. Such a model allows the dynamic assembling and re-assembling of configurations and is thus suited to deal with the problems of evolution in open systems. The following features are required: agent creation and termination, data space creation and deletion, exposing and hiding data spaces, and attaching to and detaching from data spaces.

Creation and termination of client agents

Client agents are created in APROCO by instantiating and activating an object possessing its own thread of control. A client agent is active as long as it has not deliberately stopped its processing or its execution environment has terminated the execution. Client agents cannot be terminated by other agents. They either terminate themselves or are terminated by the execution environment, in our case the Java virtual machine.

Creation and termination of coordination agents

New coordination services can be introduced to APROCO by adding new coordination agents that implement those services. Coordination agents are very similar to the client agents of the system, except that each coordination agent has to take care of participating in the flow of information through the coordination medium that its service needs to get involved with. Coordination agents do not have any special means to control the flow of forms through a specific data space. As shown in section 3.8.3 multiple data spaces are used to enable the coordination agents to control the traffic. If we add a new coordination agent, we must enable it to be part of the information flow through the system to be able to provide its service. This can be done in two ways:

1. **By a meta-level coordination agent and using a special protocol:**
   We introduce a coordination agent that allows another coordination agent to be inserted into the list of coordination agents that get the forms passed around between the client agents and are able to manipulate them, if required.

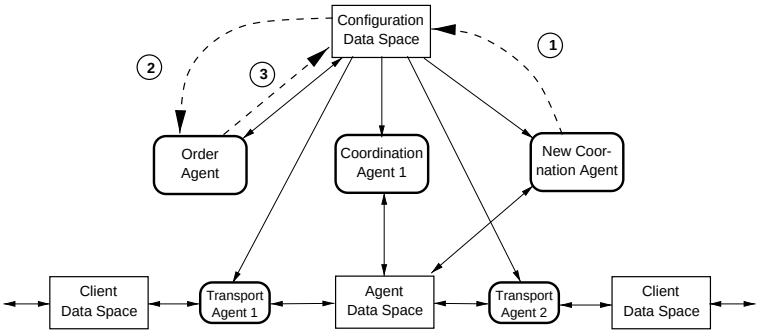


Figure 3.8: Specify the relative order of coordination agents using the order agent

In figure 3.8 the **order agent** as a meta-level coordination agent is listening to new coordination agents registering with the system (1) through the global configuration form space providing their name and the desired position in the information flow (first, last or don't care). The order agent removes the "order form" that indicates the relative order of the coordination agents from the global configuration data space (2) and adds the new coordination agent to this list and puts the new form back into the data space (3). In our example shown in figure 3.8 the relative order for forms passing the coordination medium from left to right before the introduction of the new coordination agents was e.g. `(Transport Agent 1, Coordination Agent 1, Transport Agent 2)` and after the introduction of the new coordination agent `(Transport Agent 1, Coordination Agent 1, New Coordination Agent, Transport Agent 2)`.

In order to allow this mechanism to work properly all coordination agents have to follow a specific protocol: they are only allowed to retrieve forms from the shared data space that are explicitly addressed to them, and after they performed their tasks they have to consult the "order form" in the global configuration data space maintained by the order agent to find out which coordination agent is next on the list and explicitly address the form to it.

Problems and limitations. This mechanism only allows for a flow architecture of the coordination agents, i.e. every coordination agent has a position in the list of the order agent and will get the forms addressed to it exactly once in each direction. This is only suitable for client / server kind of applications with only one type of server agents. As soon as we have more complicated interaction schemes we need other kinds of designs.

The need for a specific protocol for all the participating coordination agents is limiting its applicability. The mechanism cannot be easily added to an existing application without changing all the existing coordination agents.

2. **By dynamically reconfiguring the connections using a special interface:**
By extending the interface of the coordination agents we can achieve the ability to dynamically reconfigure the coordination medium. We can extend the coordination agents interface with a method to get all the data space references that it has at the moment and a method to set those data space references. In the implementation these methods look like this:

   - `getReferences(CoordinationAgent):[ObjectSpace]`
   - `setReferences(CoordinationAgent, [ObjectSpace])`

   In the general case the setter method can only accept an identical number of data space references as the corresponding coordination agent actually has. Without any special functionality in the coordination agent, it is only possible to reconfigure the connections between the coordination agent and the data spaces it is actually connected to. All this references are used within the coordination agent's code, and it is not guaranteed that an arbitrary coordination agent can handle more or less data space references in a useful way.

   Figure 3.9 shows how a newly created coordination agent and a newly created data space are inserted into the coordination medium. The new coordination agent needs to get a reference

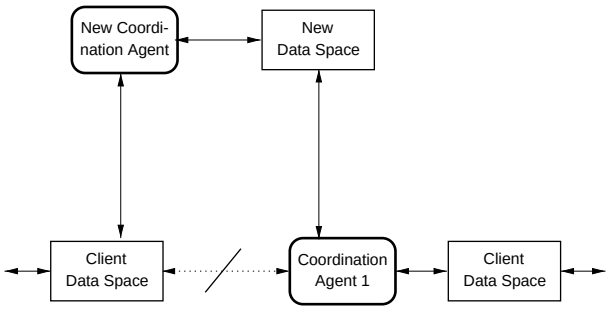


Figure 3.9: Dynamical re-configuration of coordination agents in APROCO

for the existing client data space and the existent coordination agent 1 needs to exchange its existing reference to the client data space with one to the new data space. This can be achieved by the two mentioned methods in the following way:

(a) Call the getter method of the coordination agent 1 to get the reference of the new data space.

(b) Use this reference to set the new coordination agents reference to the client data space by calling its setter method.

(c) Call the setter method of the new coordination agent with the references for the client data space and the new data space as parameters.

Problems and limitations. The dynamic reconfiguration of coordination agents can only be done by the designer of an application and not by the agents themselves, because an agent cannot directly manipulate another agent via method invocations.

Creation and deletion of data spaces

Agent configurations are built around data spaces. Every agent can create new data spaces at any time. Every newly created data space is initially a private data space only accessible to its creator. To enable other agents - including the coordination agents - to get access to a data space it must be exposed to them.

The deletion of a data spaces would only be allowed, if no agent is possessing a reference for it any more (except for the one that wants to delete it, of course). Such a deletion cannot be done explicitly, because of the nature of open systems that prevents reasoning on the number of agents that may hold a reference for a given data space at a given time. A data space can be garbage collected when no agent is actually holding a reference to a data space any more and no other data space is containing an item holding a reference for this data space. Such a garbage collection is automatically done in our implementation of APROCO, because of the built-in automatic garbage collection of the programming language Java.

Exposing and hiding data spaces

Every agent needs at least access to one data space in order to participate usefully in the system. This access can be obtained either statically at compile-time or dynamically by getting a reference to an existing data space from a well-known place - the globally accessible configuration data space. Every agent that likes a private data space to be accessible to other agents can insert a reference to it wrapped into a form into the global configuration data space as any other form. In APROCO the data space references are first class objects that can be (wrapped into forms) passed around like any other data space items. In open systems this can easily lead to problems with dangling references. We could have introduced special constructs like the *data space logicals* used with Objective Linda [Kie97], which are additional wrappers for the real references into independent representations. With this solution, it is necessary to declare another operation that transforms the logicals back into valid references for an agent to be usable. Because we limited this work to the core part of a coordination medium, and because of the time limitations, we decided to keep the medium as simple as possible and did without such a special construct. A commercial system surely has to incorporate a similar solution to this problem, e.g. by using Objective Linda as the base system to build up the coordination medium.

If a data space reference is not meant to be used by other agents any more it is possible to remove it from the global configuration data space to prevent newly created agents from getting access to it. This can be seen as an operation to hide this data space, but only for newly created agents. All the agents that already obtained the handle before cannot be forced to surrender it any more.

Attaching to and detaching from data spaces

To be able to attach to an existing data space is a core feature of dynamic composition of agent configurations. In APROCO an agent can attach to a data space by obtaining a data space reference for it through the global configuration data space.

To detach from a data space does not need any particular action. Because we do not provide the means for explicitly deleting a data space as mentioned before, we do not need to know if there are still agents around with references to a particular data space. However, for the automatic garbage collection offered by Java to work properly within APROCO, it is necessary for agents that stopped to use a particular data space to explicitly give up their references to it. This is done in Java by assigning the constant `null` to the variable that denotes this specific data space reference.

3.8.3 Multiple Data Spaces

For APROCO we employ multiple data spaces instead of just a single one as with the standard Linda model. Several approaches have been presented to extend standard Linda with multiple data spaces (see e.g. [Kie97] for a discussion). We shortly discuss the reasons for our decision to use multiple data spaces in APROCO.

No special communication required. The coordination agents in APROCO need a way to control the flow of information between client agents in order to provide their coordination services.

This can either be done with the introduction of a special type of connection to a single shared data space to enable a higher priority for the coordination agents (e.g. by generating an event when something happens in the shared data space), or by introducing multiple shared data spaces as shown in figure 3.10. The use of multiple shared data spaces enable us to use only one type of communication throughout the whole application, namely generative communication.

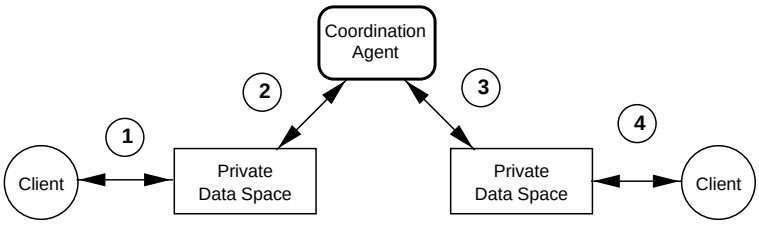


Figure 3.10: Use of multiple shared data spaces

Figure 3.10 shows a very simple situation where the two client agents operate on private data spaces. To make this system work, a coordination agent has to "connect" these two private data spaces together by transporting the forms from one data space into the other. By doing this, the coordination agent simulates the behavior of a single shared data space for the client agents.

Consider the client agent on the left side in figure 3.10 wants to send a form to the client agent on the right side. It just puts the form into the only data space that is available for it (1) and possibly waits for an answer to show up in this data space. The coordination agent is grabbing the form from the private data space of the left client agent (2) and possibly does some transformations to it to fulfill its coordination purpose and finally puts the form into the private data space of the right client agent (3). Now the client agent on the right can grab the possibly transformed form from its private data space (4) and do whatever it likes with it.

The simplest coordination agent in APROCO is the one that simply transports the forms from one data space into the other without adding or changing anything.

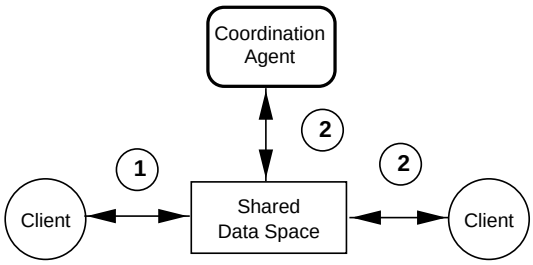


Figure 3.11: The coordination agent would need a special type of connection to the shared data space to be operational.

To compare this, we show a situation with only one shared data space in figure 3.11. If in this situation the client agent on the left wants to send something to the client agent on the right, it equally puts the form into the only accessible shared data space for it (1). But now, without any special type of connection, the coordination agent has no priority in grabbing the form before the

client agent on the right gets it (2). Both agents wants to grab (or at least read) the item but only one will get it. If it is the coordination agent that gets the form, it can fulfill its task, but if it is the client agent, the coordination service most probably will fail.

Large Systems. When the model has to deal with large systems, it is vital to be able to divide the overall configuration into smaller *subconfigurations.* This means that it must be possible to treat entire configurations like single agents at a more abstract coordination level.

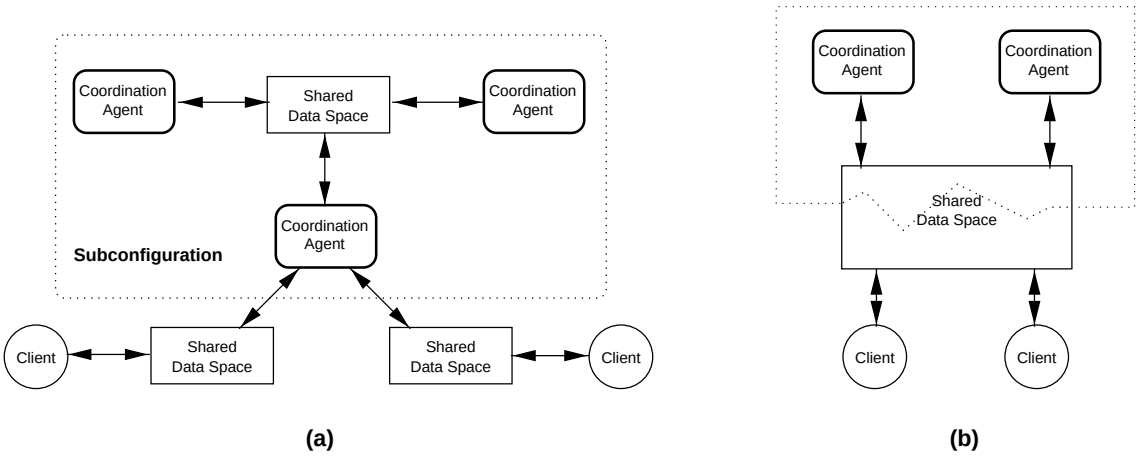


Figure 3.12: Subconfigurations can be seen as single agents (a) in APROCO, but not (b) in standard Linda.

It is not possible to view subconfigurations as single agents with a single shared data space as shown in figure 3.12. It is not possible to draw a clear border line between two subconfigurations. This leads to the necessity of introducing multiple shared data spaces into a model that is able to deal with large systems. Of course, multiple shared data spaces can be simulated using a single shared data space by logical partitioning the single shared data space. This can be done with a special object that every data space item has to carry to mark to which logical partition of the shared data space it belongs to. However, the agents' access to such marked data space items must be managed by convention, thus this mechanism cannot solve security problems as shown in the next paragraph.

Security and Naming. Multiple data spaces are important for security and naming [BCG97]. For security reasons, it is necessary to allow agents or collections of agents to use their own data space "sandboxes". It is easier to provide an untrusted agent with its own data space than try to restrict its access to a shared data space on a level needed for security issues without preventing the agent from being operational at all.

Naming is necessary if we want to provide each agent with its own name space, but still let it inter-operate with other agents. This can be solved elegantly using multiple data spaces that can be exchanged as a whole or can have some data space items transported from one to the other, as done in APROCO.

3.8.4 Designs Using Private Data Spaces

Private data spaces are shared data spaces that are only used by one client agent and the coordination agents inside the coordination medium but not by other client agents. In the multicast example (see figure 3.6) we used private data spaces for the notification recipients. We discuss the advantages of designs like this and show the necessity of private data spaces.

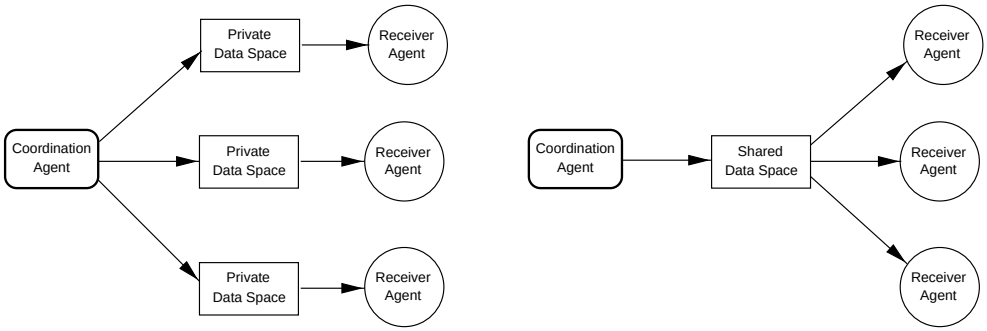


Figure 3.13: Private data spaces versus single shared data space

In figure 3.13 we show a part of the multicast example presented earlier using private data spaces and for comparison, using a single shared data space for all the recipients. The solution using a single shared data space employs a standard protocol in standard Linda systems (see e.g. [MU97]) to enable the recipients to proceed independently: the data space items are explicitly addressed to the client that should get them and all the clients are only allowed to remove those items that are explicitly addressed to them. This solution is very often used, but very inflexible in terms of client protocol. If only one client agent is not following the protocol and consumes items that are not explicitly addressed to it, the application behavior cannot be guaranteed any more. The solution based on private data spaces is much less dependent on the client protocol and allows the client agents to be as independent as possible. We list some reasons for introducing private data spaces as a design choice in APROCO.

Security. The main reason for introducing private data spaces as a design choice is to ensure security. Using private data spaces, we can guarantee that no client agent can interfere with the communication of another client agent. This is guaranteed by the fact, that no other client agent can get a reference to one agent's private data space and a reference is required for getting access to a data space.

Of course, even with private data spaces we cannot prevent a client agent from publicizing a reference to its private data space in a place that is accessible to other client agents and thus let them attach to it. This means that the mechanism of private data spaces alone cannot fully guarantee security. As we will discuss in section 3.8.5, access rights on data spaces are also required for this purpose.

Liveness. With private data spaces it is possible to create settings where guarantees for liveness of all client agents can be given, supposing the coordination medium works correctly. In this case

every client agent gets the right forms delivered to its private data space where no other client agent has access to.

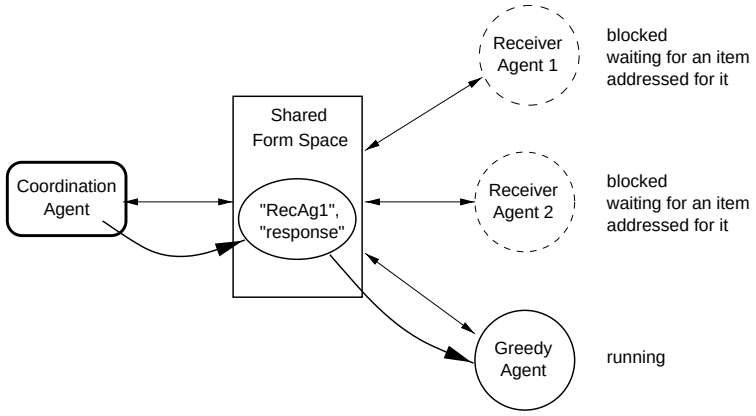


Figure 3.14: Liveness problem with single shared data space

To illustrate possible liveness problems using a single shared data space we show a situation in figure 3.14, where all client agents are using the same shared data space for their communication. In this case, it is possible, that one “speedy” and “greedy” client agent can always overtake the meant recipient of a form and grab it before the recipient agent gets woken up. If we suppose, that the implementation of the operations is not preventing one process from being indefinitely bypassed, we cannot give a guarantee for the liveness of this client agent.

If instead we use a solution with a private data space for each of the receiver agents in figure 3.14, liveness of all receiver agents can be guaranteed, because no other client agent is able to access the private data space of a particular receiver agent and “steal” its forms. Thus liveness considerations are strongly coupled with the security aspects we discussed above. In particular, liveness can only be guaranteed if security can be guaranteed, and this is only possible in conjunction with access rights on data spaces, as we will discuss in section 3.8.5.

Atomicity. Private data spaces are an easy solution for the problem of interference of agents trying to consume multiple data space items. As mentioned earlier, it is not possible in Linda and most of its derived systems to atomically retrieve more than one data space item[5]. We can imaging situations, where this behavior can lead into trouble.

If a client agent needs to read out all the items in a shared data space, it has to repeatedly consume one item after the other until there are no more items left. This is only possible, if the client agent can be sure that there is no other client agent interfering with this action and e.g. trying to do the same at the same time. Private data spaces are an easy solution for this problem, because every client agent can be sure to be the only client that has access to all the items in the data space and thus, it can read them out without being disturbed by other clients.

[5] An exception is e.g. Objective Linda [Kie97], where it is even possible to specify the minimal and maximal number of items that will be read or retrieved simultaneously in order to succeed.

3.8.5 Access Rights on Data Spaces

An agent can get access to a shared data space either statically at compile-time through the programming language or dynamically through the acquisition of a reference to the data space at runtime. In APROCO data space references are wrapped into forms that can be exchanged through another data space like any other forms.

Access rights on data spaces are an essential part of APROCO, because different access rights are necessary to implement security mechanisms in open systems.

Consider an example where different client agents are asked to give a vote for a specific issue and they are guaranteed that their votes are held secret from the other clients (see the full description of the problem and the solution in section 5.3). We have to take into account that the solution should be suited for open systems. This implies that the voting participants must be allowed to enter or leave the configuration at any time. To enable the system to service them correctly, the clients are required to register their presence with the system. Because of the reasons mentioned in section 3.8.4, an optimal solution should incorporate a private data space for each client to put its votes into. Private data spaces have special access rights: they are only accessible for the client that created the data space and all the coordination agents of the system. The coordination agents do not have a special way to access a shared data space, this means that a new client has to provide the system with a reference to its private data space in order to have successful communication with the system and thus with the other clients. This reference must be put into a place where all the interested coordination agents can get it, but a potential cheater among the clients is not allowed to grab it from there. Hence, a solution that really ensures security needs private data spaces for the clients and a registration data space (possibly the global configuration data space) with **write-only access** for all the clients.

There are only a few products around that supports special access rights on shared data spaces, one is Paradise [Sci94] from Scientific Computing Associates and another one is called T Spaces [IBM98a] and is an ongoing research project at IBM. Paradise provides multiple tuple spaces with different degrees of visibility and degrees of access to them. Tuple spaces are either private, accessible only within a single application, or semi-private, accessible for related applications, or public, accessible for all applications. Every process within an application can have full access to a specific tuple space, read-only access or write-only access. The access rights are coded into tuple space references, that can be exchanged over other tuple spaces as normal tuples, as done in APROCO using forms. A commercial coordination medium is required to support access restrictions of this kind to enable security.

Deadlocks. Restrictions on access rights do not introduce new potential deadlock situations, if they are realized by encoding the respective rights into the references themselves (or in a wrapper structure for the references), and the operations are programmed in a way, that it is not possible to call them if the required access rights are not granted to the calling agent. A potential deadlock situation would be, if an agent could send a request to a data space that it did not have the appropriate rights for and waits for an answer delivered back to it. Such a situation is not possible, if the agent would be prevented from sending a request into a data space without having sufficient rights, e.g. with an error message.

3.8.6 Forms

A *form* is a set of bindings of labels (or keys) with corresponding values. The values can be of arbitrary type, even another form is a possible value. This way nested forms can be constructed.

The concept of forms we used here is based on ideas found in incrementally "grown" systems like *electronic mail* where new features (e.g. electronic signatures) are simply added to the standard mail documents using new tags. It is important to note that the introduction of such new tags do not break all the older clients that do not know them, they simply ignore those new tags and work with the known ones.

Forms as holder of information are known for a long time already. To the author's knowledge Lisp [Ste90] was the first programming language that incorporated a similar concept. They are successfully used in areas where extensions to the standard behavior are frequent. Hence, the use of forms is well-proven and wide-spread, although not in conjunction with Linda-like shared data spaces, where tuples are the basic means of interaction.

We used forms instead of the standard tuples as information items in the shared data spaces because forms have the following useful properties:

- **Extensibility:** Extensibility can be described as the possibility to add new functionality to an existing piece of code without affecting the previous behavior. See e.g. [LAN98] for more details. With forms it is easy to add new tags and values without breaking the existent system.
- **Default values:** In the scripting language Python [vR97] (as with Lisp [Ste90]) it is possible to create functions that take their arguments by keyword - instead of by position, as with tuple based environments - and can have default values that are used when no value is given. The Python dictionaries used for this are very close to our notion of forms.
- **Easy to wrap up code:** As shown in the markup language HTML used for documents published over the web, it is easy to wrap up active behavior like small Java programs - called applets - into the forms used for the formatting of the documents. This works by introducing new tags for the code fragments that are meant to be processed by the HTML browser.

Pattern Matching

Pattern matching is the basic mechanism used in Linda based shared data space systems. It is used to compare the items in a data space with a pattern item to find one that is similar enough to be considered as a matching item. The `in(p)` and `read(p)` primitives use a pattern matching mechanism to inspect the data space and - in case of `in(p)` - to remove a matching item after its values has been read in.

Tuple matching. A tuple is a series of typed values, e.g. `("a string", 17)`. A pattern tuple is a series of typed fields, some are values, others are typed place-holders, indicated usually through a prefixed question mark, e.g. `(?s, ?i)`, where `s` and `i` are variables declared of type string and integer respectively.

A tuple *t* and a pattern tuple *p* match if:

- both *t* and *p* have the same arity (same amount of fields),
- values in corresponding fields are identical, and
- a typed place-holder in *p* and a corresponding value in *t* have the same type.

As a result of a successful matching operation, the place-holders in the pattern tuple *p* obtain the values contained in the corresponding fields of tuple *t*.

Form matching. A pattern form is a normal (possibly empty) form containing only labels and values, but no place-holders. We chose the following definition for our matching operation for forms:

Definition 1 *A form* f *and a pattern form* p *match if and only if:*

- *the labels of* p *are a subset of the labels of* f*,*
- *the values of corresponding labels are identical.*

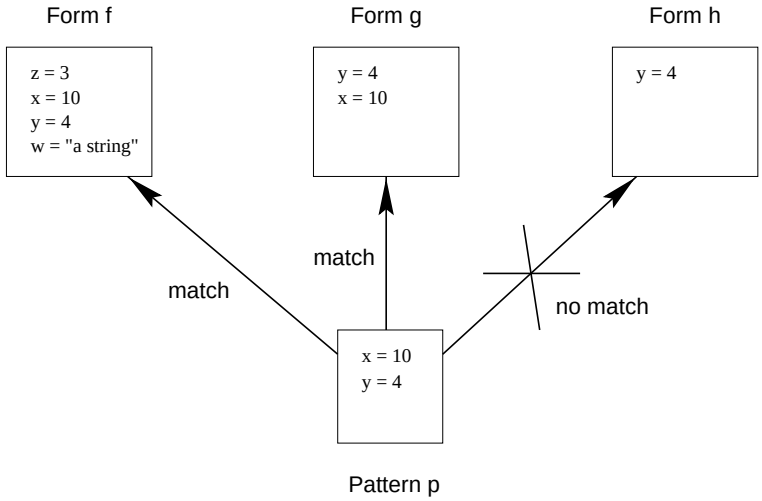


Figure 3.15: The pattern form p matches forms f and g, but not form h.

Figure 3.15 shows an example of a pattern form and what forms this pattern form can match. The chosen matching operator has the following properties:

- The order of the labels is irrelevant.
- Additional labels in the forms are ignored.
- The empty form as pattern form matches every form.
- There is no notion of place-holders and thus no assignment of values to place-holders as a result of a successful matching operation.

Pattern matching with forms is based on the pattern matching with tuples as introduced in the preceding paragraph. Tuples can be transformed into forms without losing information by creating bindings of form labels and values using form labels according to the position of the field in the tuple that the value is taken from. Place-holders in pattern tuples are just skipped and the respective position form label is not used in the transformed form. Tuples can be seen as restricted forms, therefore a matching operator for forms should be compatible to the one for tuples, in the sense that two matching tuples extended to forms should still be matching.

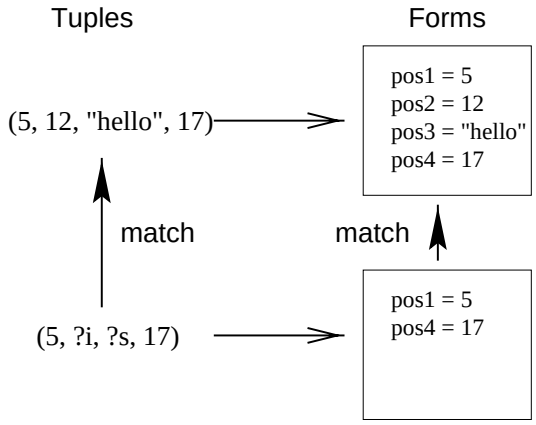


Figure 3.16: Transformation of tuples into forms

In figure 3.16 we show a possible transformation of tuples into forms and the compatibility of the used form matching operation with the tuple matching operation with an example.

Extensibility. Tuple matching is based on the check of the arity of the tuples in the first place. Only if the arity is equal the other similarities are taken into consideration. It is exactly this behavior that prevents tuple based systems from being easily extensible. We understand extensibility as the possibility to add new functionality to an existing piece of code without affecting the previous behavior.

We can see that extensibility is fully supported by the matching operation we used for forms. The pattern form is not specifying the “arity” of the target form or the type of a specific field of it. Only identical values of identical labels are considered as matching. New functionality can be added by simply introducing new labels into the forms that are passed around. A form with additional labels is still matched by a pattern form that would match the same form without the additional labels. This way, existing behavior will not be affected by introducing new functionality.

Limits. The matching operation for forms we introduced is by intention easy to understand and easy to implement. As usual, this simplicity generates some limits. With the presented form matching operation it is not possible to directly express matching of labels without their corresponding values, e.g. if we want to match a form that has a particular label, but are not interested in the corresponding value, we cannot provide an appropriate pattern form to match it with the presented matching operation. The matching operation could be extended to allow this, but we

could not find an application where we really needed this feature. We know that it would help to clarify the interface descriptions of the agents as used in the examples chapter, because it would allow to clearly state the expected keys of a particular form that will be matched, independent of the values that are associated with this keys. For the sake of simplicity of the matching operation, we sacrificed the extended clarity of the interface descriptions.

3.8.7 Form Example

This small example shows the kind of extensibility we gain in using forms as basic information containers.

Problem Description

In a client/server setting there is often the situation that a newly introduced client generation wants to have new requests serviced by the server that the old generation did not know about. Suppose that it is not possible to update all the old clients to the new version, so there will be still old versions of the client running and expecting the server to respond to their service requests.

Solution

With forms the approach to solve this very common situation is very easy and straight-forward. The server is still listening to the same interface definition, namely a form, and the clients are using the same interface as well. The new client generation may introduce new keys and values in the form as it likes to without disturbing the old clients in their operation. The server is responding to requests from the old clients the same way as from new clients. If there is no corresponding label for a new service in the request form, the server simply replies with a default value. New clients understand these new labels and values while older ones just ignore them, because they are unknown to them.

Figure 3.17 shows a situation with a server agent and two client agents of different generations using forms. The new client requests an extension to the original service while the older client only needs the original service. Using forms there are no changes needed to the older client; it simply ignores the new labels in the answer from the server. In the situation shown in figure 3.17 the answer from the server is directed to the new client (with ClientID 15), but the same answer is also accepted by the older client if it is directed to its ClientID.

Using standard tuples, either the server needs to be extended with a thread that is listening to the tuples from the old service (because the arity of the tuples are not the same any more) or the older client needs to be changed (to the new arity of the tuples), and this is not possible in our example, because the source code is not available.

Discussion

The usual approach to solve the problem is to declare a new interface for the server for the new client generation while retaining the old interface for the old clients, as had to be done for the tuple based solution. This way the server has to maintain different interfaces and react differently on requests matching one of those interfaces.

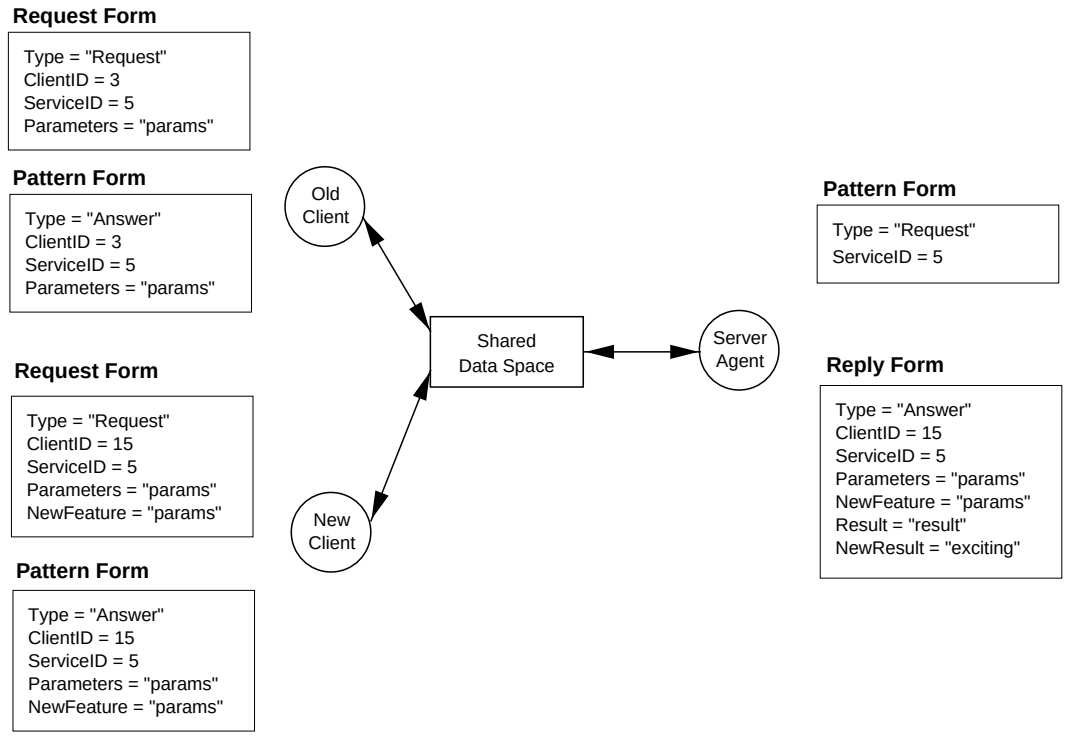


Figure 3.17: An example using forms: server with different generations of clients

Using forms the maintenance of different generations of clients is very easy and straightforward, because older clients simply ignore the parts of the reply that they do not understand. They are always able to get the reply from the server, because it is still a form, no matter how much labels are in it. It is also easier to maintain the server, because there is only one interface to take care of.

3.9 Properties

To summarize, the coordination medium APROCO has the following properties:

Flexibility. The concentration and encapsulation of the coordination activities into the coordination agents inside the coordination medium yields the possibility to adapt it to changed requirements or environments without affecting the rest of the system. It is easier to change or exchange a single component encapsulating the behavior than to change all the participants to realize another cycle in the evolution of requirements.

Ease of Reuse. The explicit representation of coordination services - abstracted away from the concrete situation and participants - allows one to reuse such independent entities in new settings easily.

Decoupling. The use of generative communication as the basic means for all agents to communicate and coordinate allows a high degree of decoupling of producers and consumers of data. This decoupling is a prerequisite for the clean encapsulation of the coordination abstractions and thus for the flexibility and reusability of them.

Independence Of Data. The use of shared data spaces allows data to live independently from the agent that produced it.

Access Rights On Data Spaces. To enable security, it is vital to have the possibility to declare different access rights on shared data spaces. Because APROCO supports multiple data spaces, it is sufficient to be able to define access rights per data space and, because of its agent architecture, per agent.

Private Data Spaces. Private data spaces allows one to create designs with guarantees for security, liveness, and maximal independence of the participating client agents very easily.

Extensibility. The use of forms as the carriers of information enables easy extensibility of the applications built with APROCO. It is possible to add new functionality without necessarily changing the existent agents interfaces.

3.10 Evaluation of The Approach

In this chapter we presented the coordination medium APROCO for software agents based on multiple shared data spaces and generative communication.

3.10.1 APROCO Fulfills its Requirements

APROCO is a programmable coordination medium for software agents with an open and dynamically changeable architecture. Because of the dynamic re-composability and decoupling of the agents it is particularly suited for open systems. APROCO allows the user to factor out the coordination behavior of an application into explicit entities, namely coordination agents. These entities for the coordination abstractions are adaptable to changed requirements and reusable in different settings.

Separation of Concerns. APROCO supports the clean separation of concerns. The client agents are mainly concerned with the computational aspect of the application - besides the communication with other client agents necessary to reach the overall goal of the application - and the coordination agents are solely concerned with the coordination aspects of the application.

Coordination Agents as Explicit Coordination Abstractions. APROCO factors out the coordination solutions into explicit, dynamically changeable entities, the coordination agents. This increases or even enables reusability and flexibility of the used coordination abstractions. The coordination agents can be combined to deliver high-level coordination services to the client agents.

Reusability of Coordination Agents. Coordination agents are explicit representations of coordination services, abstracted away from the concrete situation and participants. The level of abstraction achieved this way allows one to easily reuse these coordination agents in different settings. We will demonstrate this by reusing some coordination agents in different examples.

Flexibility of Coordination Agents. Explicit self-contained representations are easy to exchange with new ones to keep up with changing requirements. Flexibility in APROCO is not limited to this kind of exchangeability of coordination agents, the coordination agents can be realized to be flexible themselves. We show an example (the administrator / worker example in section 5.4) where the actual policy for an action performed by a coordination agent can be dynamically changed by simply replacing a form in the global configuration data space. This way we achieve run-time flexibility of our coordination solutions.

Decoupling of Agents. The use of generative communication as the basic means for all communication among the agents delivers a high degree of decoupling of producers and potential consumers of data. This decoupling is an important feature in the domain of open systems as described in section 3.4.1. The decoupling of the agents inside the coordination medium is important for the design of explicit coordination abstractions.

Dynamic Re-composition. The loose coupling of the agents enable them to dynamically join or leave a configuration without leaving others with broken communication channels. Together with the possibility for the agents to create new data spaces and attach to existing data spaces this yields the means for dynamic composition and re-composition of applications as described in section 3.8.2. Open systems cannot always be stopped and recompiled to incorporate modifications. This implies that it must be possible to modify the system dynamically. To be able to tackle the changing requirements in open systems, dynamic re-composition must be supported by any serious candidate architecture in the domain of open systems.

3.10.2 Limitations

APROCO is based on the inspection of the data flow between the client agents to fulfill its coordination purposes. This causes some limitations that are inherent to the data-oriented coordination model and cannot be changed without leaving the generative communication model. The coordination agents inside APROCO have no possibility to directly query or modify internal state of the client agents. They only have the information that is passed between the client agents. This limits the capabilities of the actions that a coordination agent can perform to those generally available in data-oriented coordination models [PA98]. We decided not to introduce special capabilites that would leave the model, because all agents can use the same communication mechanism, and thus the decoupling of the agents is higher, promising a higher degree of reusability.

Besides this inherent limitation APROCO has also some limitations due to some design decisions we made. We list this limitations here.

Conflicting Coordination Services. There is no support in APROCO to find and resolve problems arising from conflicting coordination services. If one coordination agent e.g. removes an item that another coordination agent has introduced to perform its service, this service has to fail. To resolve such kind of problems there one would have to find a way to reason about the behavior of coordination agents and to check for violations.

Chapter 4

Implementation

We implemented APROCO and some examples to experiment with the concepts and find possible limits. All the material presented in this thesis is freely available at the author's web page (URL: `http://www.iam.unibe.ch/~dkuehni/aproco.html`). In the following sections we discuss some implementation issues.

4.1 Implementation Overview

For the implementation of the coordination medium APROCO we used the object-oriented programming language Java [Fla97] together with an implementation of the Linda primitives in Java, called Jada ("Java Linda") [CR96]. Jada offers a set of Java classes for creating multiple shared data spaces and operating the Linda primitives `in`, `read`, and `out` on them. The agents used in APROCO are implemented using Java *threads* that can use those primitives to generate or consume data space items, or inspect data spaces associatively for data space items. In our implementation all Java threads run within a single Java virtual machine, this means that there is no physical distribution in our implementation yet. For convenience we made use of a freely available class library for Java, called JGL [Obj97]. This library offers a bunch of highly customizable object containers and algorithms for Java. We used it to implement the forms in APROCO, thus all actions related with forms need JGL to be imported. We describe our form implementation in detail in appendix A.
We introduce the different parts of our implementation in the following subsections.

4.1.1 Java

All agents in APROCO are built with their own Java thread running in the same Java virtual machine. We used Java in this thesis for several reasons. First of all, the programming language chosen to build a coordination medium for open systems must be suitable for open systems, i.e. it must be running on different platforms and support distribution. Java is running on a variety of hardware and software platforms and its wide adoption guarantees for even quite exotic platforms to be available soon. Java supports distribution by offering standard class libraries for sockets and remote method invocation (RMI). The upcoming Java 1.2 will add CORBA functionality to

this basic level of distributed computing and thus will increase inter-operability. Secondly, Java is the first choice as the programming language to investigate software agents especially if they are meant to be mobile, mainly because of its ability to dynamically load program code into a running application and its platform independence. By the time of writing this thesis there were a lot of Java agent frameworks available to investigate the power of this concept, IBM's Aglets SDK [IBM98b] and ObjectSpace's Voyager [Obj98] are possibly the most prominent representatives today.

Problems and Limitations. During the implementation of APROCO using concurrent Java threads that need to be synchronized, we encountered a disadvantage of Java. If several threads are running with the same priority in a Java virtual machine, fairness cannot be automatically guaranteed. Running Java threads are only preempted by threads that have a higher priority (see next section for more details). This implies that in the situation described in section 3.4.1 where two processes are protected from concurrently executing a critical section of their code by using a shared resource in Java it is possible that one thread indefinitely overtakes the other thread. To prevent this situation we had to explicitly yield the processor to other threads using `Thread.yield()` at appropriate locations in the code.

Although this problem is known and documented (see e.g. [Lea97]) and thus a "feature" instead of a bug, it is nevertheless a very annoying feature.

4.1.2 Jada

Jada is a Java class library designed for the use in distributed applications for the World-Wide-Web (WWW). Jada can create multiple shared data spaces (so called *Object Spaces*) that can be accessed remotely. An Object Space holds Java objects as data space items. In order to allow objects of user-implemented Java classes to be part of data space items, they have to implement a special Java interface called *JadaItem*. This interface defines methods for dumping objects to byte streams and constructing objects out of byte streams for the remote access of Object Spaces over IP connections, and a matching predicate.

Jada is freely available for research projects and can be downloaded from its creators home page (URL: `http://www.cs.unibo.it/~rossi/jada/index.html`). Jada is written using Java 1.0.2, but we encountered no problems running it with Java 1.1.

We present a detailed description and discussion of Jada and some of the implementation details in appendix A.

Problems and Limitations.

Access Rights on Data Spaces. Jada offers no means for restricting access to a data space or the items in a data space. As we discussed in section 3.8.5 access rights on data spaces are essential for a coordination medium to be able to offer security for the client agents.

Fairness. Jada is multi-threaded and offers blocking of threads or remote applications over its Object Spaces on the level of single threads. However, Jada uses the thread management capabilities of standard Java, and they are quite poor. Because Java is designed to be platform independent,

the requirements for the Java virtual machine have to be defined in a way that they can be met on every platform. For the scheduling of threads this means that fairness even in the weakest sense cannot be guaranteed [Lea97]. Running threads are only preempted by threads running with a higher priority. In our experiments we found out that one has to explicitly yield the processor to another thread to give it a chance to grab a shared resource in a Jada Object Space. This can be done by explicitly calling `Thread.yield()` after the shared resource has been released and before another try to get hold on it is started.

4.2 Interface Notation and Implementation

Since all agents in APROCO exclusively use generative communication to interact with their environment, an external interface only has to list the kind of forms that are put into a specific shared data space and the templates for the pattern form used to read out or consume forms from that specific shared data space. For every data space the agent is connected to we list a separate table. Each table consists of three parts:

1. the **creates** section lists all forms that this agent writes in the given data space using an `out` operation. For each different type of forms that the agent can create we list the used keys and some example values.

2. the **reads** section lists all forms that this agent can read from the given data space using a `read` operation. Because this operation needs a pattern form as parameter, which is a normal form that consists only of those key/value pairs that at least have to be present in a form to be matched and thus read out from the data space, we only list the keys and values from this pattern form. The concrete form that is read out from with the corresponding `read` operation typically has much more keys and values as listed in this section. We list each different pattern form that this agent uses to match forms in the given data space.

3. the **consumes** section lists all forms that this agent consumes from the given data space using an `in` operation. Like in the reads section, we only list the pattern forms that are used by this agent to match forms in the given data space. This means that the matched forms can have much more keys and values as we list here, because as mentioned above, the pattern form only consists of the minimal set of keys and values that have to be present in a form to be matched and thus consumed from the given data space.

If an agent does not use a particular operation (`out`, `read`, or `in`), the corresponding part of the table is omitted.

Figure 4.1 shows an example of a typical interface of a client agent in APROCO. The reads part of the table is omitted because this client does not use a `read` operation in its code.

Note that we use example values rather than types in all interface descriptions, because the matching operation we use for forms (defined in section 3.8.6 and implemented as shown in section A.5) only checks for exact match of values and does not allow to match all items of a specific type. Although this could be achieved by an appropriate matching operations there was no need for it in the examples that we show in chapter 5.

Name of the agent: Name of the data space

Keys	Example Values	Comments
	creates	
`Type`	`"Request"`	Type of the form.
`ClientID`	`5`	The client's ID.
`ServiceID`	`17`	The ID of the service.
`Parameters`	`"params"`	The parameters for the service.
	consumes	
`Type`	`"Answer"`	Type of the form.
`ClientID`	`5`	The client's ID.
`ServiceID`	`17`	The ID of the service.

Figure 4.1: Example of a typical interface of a client agent in APROCO

The Implementation

We present the Java code for the interface of the typical client agent as shown in figure 4.1.

```
ObjectSpace os = new ObjectSpace();

Form f = new Form();
f.put("Type", "Request");
f.put("ClientID", new Integer(5));
f.put("ServiceID", new Integer(17));
f.put("Parameters", "params");

os.out(new Tuple(f));
```

```
Type        "Request"
ClientID    5
ServiceID   17
Parameters  "params"
```

Figure 4.2: The creates part of the interface and its implementation

Figure 4.2 shows the implementation of the creates section of the interface of the typical client agent and figure 4.3 shows its consumes section.

In figure 4.3 the Form object called *f* in the code describes a pattern form used to match a form in the data space that contains at least the given keys and values, but of course also some results that we are interested in. The desired information can then be read out of the resulting form using the `get` method of the `HashMap` class of the JGL library that we used for the implementation of our Form class (see section A.5 for a description and the code). We present the standard idiom used in APROCO to retrieve such information.

To access a Form object in a shared data space one has to follow a special idiom shown in figure 4.4. First a pattern Tuple object (the `Tuple` class is part of the Jada package, we describe it in section A.2.1) is created containing the desired pattern Form object (called `patternForm` here) using `new Tuple(patternForm)` (1). This Tuple object is provided to the `in` operation as pattern to match an appropriate object in the desired data space (called `os` here). The actual matching is performed in two steps:

```
ObjectSpace os = new ObjectSpace();

Form f = new Form();
f.put("Type", "Answer");
f.put("ClientID", new Integer(5));
f.put("ServiceID", new Integer(17));

Tuple t = (Tuple)os.in(new Tuple(f));
Form result = (Form)t.getItem(0);
```

```
Type       "Answer"
ClientID   5
ServiceID  17
```

Figure 4.3: The consumes part of the interface and its implementation

```
(1) Tuple  tuple  = (Tuple)os.in(new Tuple(patternForm));
(2) Form   form   = (Form)tuple.getItem(0);
(3) String result = (String)form.get("Result");
```

Figure 4.4: Standard idiom to access information from a data space

1. the standard Tuple matching defined in the Jada Tuple class is used to find Tuple objects that hold exactly one Form object, and
2. the matching predicate `matchesItem` of this Form object is called to find out, whether a matching Form object exists. If this Form object contains other objects that support their own matching predicates, they are called recursively.

The resulting object of the `in` operation is a Jada Tuple object. To retrieve the Form object wrapped in this Tuple object, we have to call a special method `getItem(FieldPostion: int): Object` on the Tuple object with the position of the desired tuple field as parameter. Since the Form object is the only object that the Tuple object contains, it is the first tuple field (2). The resulting object is now the Form object that we are interested in. To retrieve the values of some keys inside this Form object, we call the `get(Key: Object): Object` method on the Form object with the desired key as parameter and get the value, if present; `null` otherwise.

4.3 Recipe for Building an APROCO Agent

As described in section 3.3 a typical client agent in APROCO performs the following tasks:

- **create** new private data spaces,
- **expose** its own created data spaces for other clients to attach to them,
- **attach** itself to existing data spaces, and
- **produce** (with `out`), **read** (with `read`), and **consume** (with `in`) forms from the data spaces it has got references for - either statically through the programming language or dynamically acquired by attaching to them.

A typical coordination agent (see section 3.4.3 for details) can perform the same tasks, but typically does not create private data spaces and thus does not need to expose them to other agents. The distinction between client and coordination agents is not important in terms of implementation, it has only been introduced for conceptual reasons.

In chapter B we show the complete code of some coordination agents and one complete example.

Creating Data Spaces

An agent can create as many data spaces as it wants to by creating instances of the `ObjectSpace` class defined in the Jada package (see chapter A for a detailed description of Jada). Figure 4.5 shows the Java code needed for this task. Please note that the location of the Jada package must be present in the `CLASSPATH` environment variable to be able to use it. Future versions of Java may solve this differently.

```
import jada.*;

ObjectSpace dataspace = new ObjectSpace();
```

Figure 4.5: Creation of data spaces

Exposing Data Spaces

To be accessible for other agents (client as well as coordination agents) a newly created data space needs to be exposed to them to let them attach to it. This is done by wrapping a reference to this data space into a form and putting this form with an `out` operation into a well-known data space representing a global environment, the global configuration data space (see section 3.4.1 for details). Every agent gets a reference to the global configuration data space by default at compile-time. Figure 4.6 shows the Java code needed for this task.

```
import jada.*;
import aproco.lib.Form;

void registerAsNewClient() {
  Form f = new Form();
  f.put("Type", "RegisterClient");
  f.put("Handle", dataspace);                    // The reference to the created data space
  globalspace.out(new Tuple(f));
}
```

Figure 4.6: Exposing data spaces for the use by other agents

In this example the client agent registers with the medium using the shown idiom and makes use of the presence of a *registration agent* (see section 3.5 for a description) that collects the references of the client agents for the use by the coordination agents of the medium. However, client agents are free to expose their data spaces (as long as they should not be private) using non-standard forms with user-defined arguments.

Attaching to Data Spaces

If an agent needs to get access to a data space in order to put an item into it or inspect the data space for a specific item it needs to get a reference for this data space. If the agent did not get a reference statically at compile-time, it can try to get one dynamically. This is done with an `in` or `read` operation on the global configuration data space (see section 3.4.1 for details)to get a form with the reference for the desired data space. If the data space was created by another client, it has first to be exposed before the agent can attach to it. In figure 4.7 we show the Java code to perform this task. It makes use of the standard idiom to access information from a form stored in a data space as described in the preceding section and shown in figure 4.4.

```
import jada.*;
import aproco.lib.Form;

registerClient = new Form();
registerClient.put("Type", "RegisterClient");

Tuple t = (Tuple)globalspace.in(new Tuple(registerClient));
Form f = (Form)t.getItem(0);
ObjectSpace handle = (ObjectSpace)f.get("Handle");
```

Figure 4.7: Attaching to a data space

Figure 4.7 uses the same naming conventions for the form containing the data space reference as used in figure 4.6.

Once an agent has got a reference for a data space it can perform the usual primitive operations on it.

This is a code excerpt of the registration agent mentioned above. This agent collects the data space references of the client agents to be used by the other coordination agents of the medium. The full code of the registration agent can be found in section B.1.3.

4.4 Polling Versus Internal Threads

A coordination agent that needs to inspect several data spaces or the same data space for different types of forms (using different pattern forms) needs to do this in an efficient way. If it would just use one blocking `read` or `in` operation at a time, this would result in the agent being prevented from grabbing other forms in possibly different data spaces.

```
while (true) {
  for (Enumeration e = clientInfo.elements(); e.hasMoreElements();) {
    ObjectSpace handle = (ObjectSpace)e.nextElement();// Data space reference.
    t = (Tuple)handle.in_nb(new Tuple(patternForm)); // Check this data space
    if (t != null) {                                 // If a form is matched
      Form f = (Form)t.getItem(0);
      agentspace.out(new Tuple(f));                  // it will be transported.
    }
  }
  try { Thread.sleep(200); }                         // Waits for 200 milliseconds.
  catch (InterruptedException e) { };
}
```

Figure 4.8: Polling style using non-blocking operations

One solution is to use only non-blocking operations to inspect the data spaces and loop until the required forms are available as shown in figure 4.8. This of course is not the best solution in terms of system resources, but sometimes the only one that is really applicable. In a situation where a coordination agent has to check a potentially big number of data spaces for forms it is much easier to go through a list of data space references and check using non-blocking operations. In the example shown in figure 4.8 the agent uses a Java Vector with data space references (called `clientInfo`) and checks each data space stored in this Vector for forms to transport into a special data space called `agentspace`. Since Java Vector objects are growable, we cannot predict how many data space references it will eventually contain, thus the polling style is applicable here.

The other solution is to use internal threads that use themselves blocking operations for the inspection of data spaces as shown in figure 4.9. For each pattern form that a data space has to be checked for and for each data space an own thread is started with a blocking `in` or `read` operation. This is elegant and saves system resources, because there is no need for constantly polling any more. The example in figure 4.9 shows a part of a coordination agent that defines an member class to wait for a matching form to be present in a particular data space and to consume it using an `in` operation. For each form that the coordination agent has to wait for in this data space it starts up such a thread with the data space reference and the pattern form as parameters. The internal thread is blocked until an appropriate form is available and gets woken up and proceeds, in this example with calling a method that processes the found form.

Note that if a coordination agent has to grab all kinds of forms in a specific data space, there is a much easier solution than the one just presented. As mentioned earlier the empty form as a pattern form is matching all forms in a shared data space. Thus it is sufficient to use a single thread using a blocking `in` operation with an empty form as the parameter for this purpose.

```
private class WaitingThread implements Runnable {
                                                  // Member class for the threads.
  Form form;                                      // The pattern form to wait for.
  ObjectSpace ospace;                             // The data space to look for it.

  private WaitingThread(ObjectSpace os, Form f) {
    ospace = os;
    form = f;
  }

  public void run() {
    while (true) {
      Tuple t = (Tuple)ospace.in(new Tuple(form));
      Form g = (Form)t.getItem(0);
      doSomethingWithIt(g);                       // Call some method with the matched form.
    }
  }
}

/* In the main method of the coordination agent */
  // check for forms to multicast, start a thread for each form
  for(int i = 0; i < numForms; i++) {
    new Thread(new WaitingThread(agentspace, toListen[i])).start();
  }                                               // toListen is an array of Form objects.
```

Figure 4.9: New thread for each inspection using blocking operations

Chapter 5

Sample Applications

In the preceding chapters we outlined the coordination medium APROCO. In this chapter we present some coordination examples to demonstrate the the different coordination abstractions as coordination agents that we listed in chapter 3 embedded in an environment. We show how multiple simple coordination agents can be combined to realize higher level coordination abstractions. Some coordination agents are used in different settings to show their reusability. The examples are mainly taken from the coordination literature and are small. In coordination literature there is a consensus about the main topics that a coordination model has to deal with [GC92]: communication, synchronization, and creation of processes. In distributed systems the main themes related to coordination are [CDK94]: mutual exclusion and elections. The set of examples we present in this thesis deals with the main topics of coordination in open systems:

- Communication is a central issue in all the chosen examples. The multicast example shows a fundamental communication mechanism used in a variety of situations.
- Synchronization is a built-in property of the generative communication style used in APROCO and thus inherent in all the examples.
- Creation of processes is used in most of the examples to illustrate the standard situation in open systems where agents can join or leave a configuration at any time.
- Mutual exclusion is shown in an example using generative communication in section 3.4.1.
- Elections are used in distributed systems to assign a special role to one process among a group of processes. We present a more general application domain of this scheme in the electronic vote example.

We do not present a full-scale real wold application, but we think that the five[1] simple applications demonstrate the main properties of APROCO and can help to convince the reader of the usefulness of APROCO as a coordination medium in open systems.

Because the agents we use in the examples have a very limited functionality, we do not present their full Java code here. Instead we show their interface in terms of the primitive Linda operations

[1]There are four sample applications in this chapter and one in the preceding chapter (see section 3.7).

they use and explain their main functionality and the way the interact with the other agents of the application to realize the overall application behavior.

The following sections describe the chosen examples in detail:

1. The **fault tolerance service** example shows how easily a coordination service can be introduced into an application that is transparent for the client agents.

2. The **observer** example is a standard design pattern in object-oriented programming. We reused a coordination agent introduced in the multicast example from section 3.7.

3. The **electronic vote** example was introduced by Minsky to show his approach of coordination policies. We use it to demonstrate how a set of coordination agents together with some design decisions can be used to guarantee security.

4. The **administrator / worker** example is an architectural pattern from parallel programming and is used here to show the dynamic exchange of the used coordination policy.

5.1 Fault Tolerance Service

5.1.1 Description

This is a client/server example where the client agents ask for a service to be done and wait for the answer to be present in the shared data space. The server agent, that maintains this particular service is unstable and can fail to deliver an answer. In this case the coordination medium has to guarantee that the client gets a service answer eventually. There could be a replicated server agent waiting on the same shared data space, and because of the anonymity of communication in the generative communication style, acting in parallel with the original server agent in normal operation. In case of failure of one server agent, this replicated server agent would do the job that both server agents did before, only slower. Unfortunately this mechanism does not provide a full fault-tolerance coordination service. If one server agent goes down after it has grabbed the client agent's service request and before it could deliver the answer back no other server agent can process this request any more. A real fault-tolerance service has to take care of this situation as well.

In any case, the client agent must be able to accept an error message as a valid answer to a service request, because there is still the possibility that there is no running server agent left to process it within useful time. In this case, the client gets an error message and it is up to the client to repeat the request at a later time to see whether a server agent for the requested service is up and running again.

5.1.2 Coordination Problems

The main problem in this example is to establish a fault tolerance coordination service for the clients that must not interfere with their normal operation. This service should be as transparent as possible for all the client agents (clients and servers are clients of the coordination medium).

5.1.3 Solution

The solution (see figure 5.1) consists of two separate shared data spaces, one for the clients and one for the servers, and a coordination agent (called fault tolerance agent), that moves valid requests and answers from one data space to the other while providing the fault-tolerance coordination service to the client agents. The clients and the servers are made as if they would operate on a single shared data space in between them. Because the coordination agent has no special means to intercept the client agent's communication (as explained in section 3.4.3) the single shared data space is split up into two shared data spaces that are connected through the coordination agent transporting the forms to and fro.

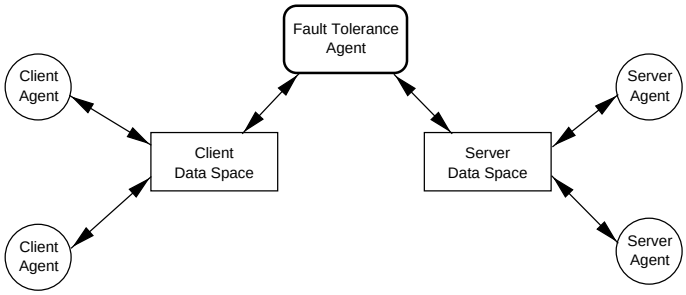


Figure 5.1: Solution for the fault tolerance service example

Client. Each client agent has a unique client ID which it adds to all its requests to enable the system (the coordination medium or the servers) to direct the answers back to the right client. The clients have to be able to accept error messages as results for service requests. They are free to maintain their own timeouts for a repetition of unanswered requests. Repeated requests due to this timeouts should be filtered out by the coordination medium not to use system resources unnecessarily.

The agents operating as clients need to have the following interface (the IDs for the client and the service are examples and may vary):

Client: Client Data Space

creates		
`Type`	`"Request"`	Type of the form.
`ClientID`	`5`	The client's ID.
`ServiceID`	`17`	The ID of the service.
`Parameters`	`"params"`	The parameters for the service as a string.
consumes		
`Type`	`"Answer"`	Type of the form.
`ClientID`	`5`	The client's ID.
`ServiceID`	`17`	The ID of the service.

As explained in section 4.2 the form in the consumes section of the table is a pattern form. This means that it does not contain all the keys and values that an expected form really contains, but the

minimal set of keys and values that it needs to be matched by the matching operation presented in section 3.8.6. In this concrete example the client of course needs the results that are contained in the form that it consumes, but since it cannot know what results to expect, these keys are simply omitted in the pattern form.

Server. Every server agent is listening for service requests directed to the service it is providing (denoted by a service ID). The servers have to include the ID of the client agent whose request they processed in the response. Otherwise it is not possible to direct the answer back to the right client agent using the design shown in figure 5.1. In this design we are using an explicit addressing style to allow the agents to get the right service requests for the server agents, and the right answers for the client agents respectively. If we would use private data spaces for the clients - as we will show in other examples - this requirement for the server agents would not be necessary.

The agents operating as servers need to have the following interface (the IDs for the service is an example):

Server: Server Data Space

creates		
`Type`	`"Answer"`	Type of the form.
`ServiceID`	`17`	The ID of this service.
`ClientID`	`5`	The requesting client's ID.
`Parameters`	`"params"`	The parameters for the service.
`Result`	`"result string"`	The result of the service as a string.
consumes		
`Type`	`"Request"`	Type of the form.
`ServiceID`	`17`	The ID of the service.

The type for the parameters and the result for this service are examples here. Since forms can contain arbitrary Java objects (as explained in section 3.4.2), we can take whatever is needed for this particular service as its parameters. In the shown example we wrapped the parameters and the result in a String object.

Note that the parameters cannot be part of the consumes section of the table, since this represents a pattern form and, again, there are only those keys and values in a pattern form that needs to be present in a form to be matched. The server agent expects the parameters to be part of the consumed forms, though, but because we have defined the matching operation for forms only to match exact values (see section 3.8.6) we cannot directly express this in the interface description.

Coordination Agents

Fault Tolerance Agent. The fault tolerance agent's job is to maintain a timeout for every service request that was sent by a client agent and to supply the client with an error message if the timeout ran out before the corresponding server agent's answer arrived. The fault tolerance agent is responsible for the transport of request forms and answer forms to the corresponding data spaces and checking for timeouts at the same time. If there is more than one server for the same service around, the special agent can repeat the request after the timeout ran out to enable another server

to respond to it. In this case the coordination agent has to pay attention to repeated requests of the client itself, due to its own timeout, and just ignore such repeated requests not to use up system resources unnecessarily. In our solution here the coordination agent does not know how many server agents are present in the system and what services they provide since this is not necessary to provide the coordination service. If a timeout for a client request runs out, the fault tolerance agent repeats the request once to enable another server (that operates the same service) to respond. After this timeout runs out again, the control agent creates an error answer for the client. In the case of a late answer to a request that already got repeated, one answer must not be transported back to the client to prevent the client from an invalid answer for a subsequent request for the same service. The timeout for this particular service should then be adjusted to prevent unnecessary repeating of requests.

The fault tolerance agent has the following interface for the two shared data spaces that it is connected to. The information in the created forms is example information and is not fixed to the shown values.

Fault Tolerance Agent: Client Data Space

creates		
`Type`	`"Answer"`	Type of the form.
`ServiceID`	`17`	The ID of this service.
`ClientID`	`5`	The requesting client's ID.
`Parameters`	`"something"`	The parameters for the service.
`Result`	`"something new"`	The result of the service.
consumes		
`Type`	`"Request"`	Type of the form.

Fault Tolerance Agent: Server Data Space

creates		
`Type`	`"Request"`	Type of the form.
`ClientID`	`5`	The client's ID.
`ServiceID`	`17`	The ID of the service.
`Parameters`	`"something"`	The parameters for the service.
consumes		
`Type`	`"Answer"`	Type of the form.

The forms that are consumed on one shared data space are created and put into the other and vice versa, because the fault tolerance agent is mainly transporting the forms back and forth. However, sometimes a form needs to be created out of the agents memory and put into the server data space, when the timeout ran out without a corresponding answer to a particular request.

The full Java code of the fault tolerance agent can be found in section B.1.2.

5.1.4 Evaluation / Discussion

In this example we showed how a fault tolerance service can be provided by decoupling the clients and servers and introducing a coordination agent with the desired functionality in between that

transports the requests and answers from one shared data space into the other. The design we used for the solution with all the servers sharing a single shared data space allows the coordination agent to repeat an unanswered service request to see whether another server agent is operating the same service and will eventually answer the repeated request. We could say that generative communication already has built-in fault tolerance, because this mechanism would work even in the case the coordination agent would only transport the forms back and forth without repeating unanswered requests. In this case, a request will be taken from the shared data space by the next server agent operating this service that is available. We would like to use this behavior, because it gives us *agenda parallelism* [CG90] without any additional effort by the coordination medium, but we cannot say that this built-in mechanism is a real fault tolerance service. Consider the situation, where one server agent just grabbed the request form and then before it could deliver the answer back into the shared data space, goes down, there is no fault tolerance in this situation any more with the built-in mechanism. In this situation our coordination agent can still provide fault tolerance correctly.

If the server agents are made to return the forms they got as requests and just add the answers they generated to them, a lot of additional functionality can easily be included, e.g. each request could be given a unique ID and a time stamp to be able to identify each request and answer pair correctly. This demonstrated the easy extensibility using forms instead of tuples. The server agents can use the special operations `merge` or `update` defined in our form implementation (see section A.6 for details).

5.2 Observer

5.2.1 Description

This is a small example derived from the observer pattern [GHJV95], a frequently used design pattern in object oriented programming. There are two types of objects involved in the observer pattern: the subject and one or more observers (often also called views). The subject can change its state and has to notify this to all the interested (registered) observer objects through some sort of multicasting protocol. After being notified about the subject's state change the observers can then request the part of the subject's state that they are interested in. The subject has to provide a method for this purpose. In the original pattern, the subject object has to provide methods for the observers to register and de-register with it to get the notifications or to stop getting them respectively. If this pattern is applied to a situation in an open system, observer agents can join and leave a configuration at any time registering only with the coordination medium, and thus the information about the actual set of interested observers is not available to the subject agent. This information is only available to the coordination agents within the coordination medium. This implies that in an open system the subject agent cannot be responsible for the multicasting of its state changes, this is the job of the coordination medium APROCO. This example reuses a coordination agent that we introduced in the multicast example in section 3.7.

5.2.2 Coordination Problems

The main problem in this example is to coordinate the state of several agents with the state change of another agent that has no knowledge of the whole configuration.

5.2.3 Solution

The solution (see figure 5.2) consists of a set of observer agents each with its private data space, one subject agent with a data space to publish the notifications about the state changes, and a different data space to accept and respond to state requests from the observer agents, and a multicast agent that reads the notifications and distributes them to all the registered observers. The observer agents have to register with the coordination medium through the global configuration data space (as described in section 3.4.1) to provide the coordination medium with a reference to their private data spaces in order to be serviced by the coordination agents. This registration information is maintained by the registration agent. Observers can dynamically join the configuration at any time and will be serviced as soon as they register with the system.

In this solution we used private data spaces in conjunction with a shared data space for all client agents. This allows us to use the advantages of both designs without having to deal with their disadvantages as discussed in section 3.8.4.

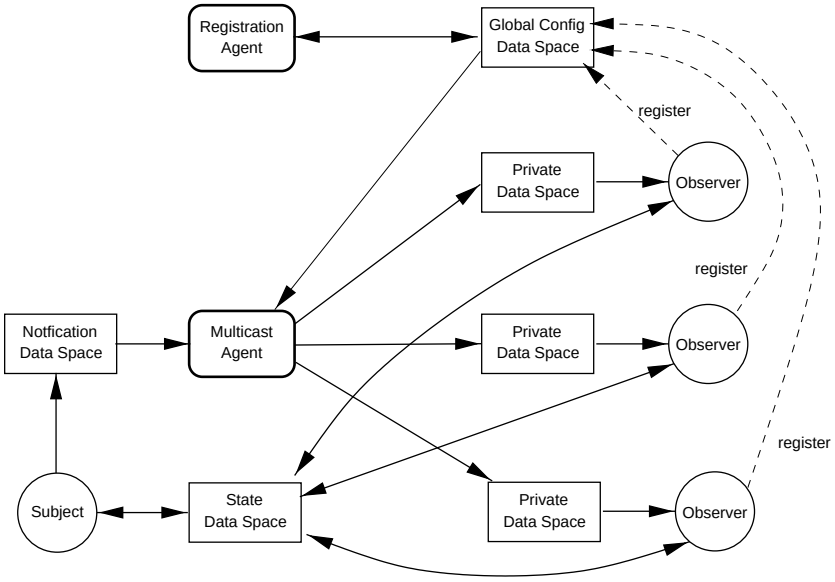


Figure 5.2: Solution for the observer example

Subject. The subject agent can change its state and has to inform some other agents about these changes. The first step is to notify those agents that something has changed. After that the subject agent waits for the notified agents to ask for parts of its actual state. The subject agent must offer some methods for this purpose. As mentioned above the subject agent in APROCO has no information about the set of agents that are interested in being notified about the state changes. For

this reason it is only generating one notification form and puts it into the multicast data space to be multicasted to the observer agents. The state requests and its answers are consumed and generated into a different data space that is shared among all client agents.

The subject agent has the following interface to the shared data spaces that it is connected to:

Subject: State Data Space

creates		
`Type`	`"Answer"`	Answer to state requests.
`ObserverID`	`5`	The recipient observer's ID.
`NewState`	`StateObject`	Represents the new state.
consumes		
`Type`	`"Request"`	Request for actual state.
`StateRequest`	`"MethodID"`	Method to be executed.
`Parameters`	`"params"`	Parameters for the method call.

The `StateRequest` is a string that identifies a method that the subject agent supports for queries of its actual state. The parameters of this method call are wrapped into a string (as an example, they could as well be wrapped into an array of objects) separated in another key/value pair of the form. The resulting object generated by this method can be wrapped as object reference into the answer form, because we are only using threads running on the same virtual machine.

Subject: Notification Data Space

creates		
`Type`	`"Notification"`	State change notification.

Observer. The observer agent is interested in getting notifications about the state changes of the subject agent. When an observer agent joins the configuration it has to register with the coordination medium by providing a reference to its private data space. After this registration it will get the notifications delivered into this private data space. To get parts of the actual state of the subject agent it puts state requests into a shared data space and consumes answers addressed to it from the same data space.

An observer agent has the following interface to the data spaces it is connected to:

Observer: Global Configuration Data Space

creates		
`Type`	`"RegisterClient"`	Registration form.
`Handle`	`ObjectSpace`	Data space reference for private data space.

Observer: Private Data Space

consumes		
`Type`	`"Notification"`	State change notification.

Observer: State Data Space

creates		
`Type`	`"Request"`	Request for actual state.
`ObserverID`	`5`	This observer's ID (example).
`StateRequest`	`"MethodID"`	Method to be executed.
`Parameters`	`"params"`	Parameters for the method call.
consumes		
`Type`	`"Answer"`	Answer to state requests.
`ObserverID`	`5`	This observer's ID (example).

See the remarks in the paragraph about the subject client for an explanation of the `StateRequest` and the resulting `StateObject` that the observer is interested in getting.

Coordination Agents

Multicast Agent. The multicast agent has been presented in section 3.7 and is reused here to solve a part of the coordination problem of this example. The multicast agent distributes forms originating from the subject client agent from the multicast data space into the private data spaces of the observer agents. The multicast agent is parameterized to allow one to specify the forms that have to be transported by it, in this case the state notifications. Because the state notifications are the only type of forms that are present in the multicast data space, the multicast agent can be told to transport all types of forms. This can be achieved by using an empty form as the pattern form.

As mentioned with the multicast example in section 3.7 the multicast agent has to update its list of interested receivers (the observer agents in this case) to be able to get the references of the private data spaces of the newly registered agents to service them properly. This update is performed each time the multicast agent gets a notification from the subject agent (or more generally the sender agent in the multicast example), and before this notification is transmitted to all the interested client agents.

The multicast agent has the following interface to the data spaces it is connected to. We only list the interface for one private data space, but the interface to all other private data spaces are identical. The symbol "*" is used to indicate that all keys and values are accepted, it can be seen as a sort of "wildcard."

Multicast Agent: Global Configuration Data Space

reads		
`Type`	`"ListOfClients"`	The list of actually registered clients.

Multicast Agent: Notification Data Space

consumes		
`*`	`*`	All forms are consumed (notifications in this case).

Multicast Agent: Private Data Spaces

creates		
`Type`	`"Notification"`	State change notification.

Registration Agent. The registration agent listens for new client agents registering their presence with the system and transforms this information into special forms for the coordination agents that need it. These interested coordination agents have to register with the registration agent, if they need to get a notification addressed to them when a new observer registered to the system. The registration agent is maintaining a form containing a list of the actually registered observer agents in the global configuration data space. If a coordination agent is only interested in getting this information (mainly the private data space reference of the observer agent), but does not need an addressed notification, there is no need to register with the registration agent, because this list is updated and available anyway. For this reason, the multicasting agent does not have to register with the registration agent to be able to get the information.

Although not used in this particular example, the registration agent must be able to handle the de-registration of client agents as well. If a client agent de-registers with the coordination medium, the registration agent has to update its list of registered client agents and has to notify the interested coordination agents about the change as well.
The registration agent has the following interface to the global configuration data space:

Registration Agent: Global Configuration Data Space

	creates	
`Type`	`"ListOfClients"`	The list of actually registered clients.
`List`	`ListOfClients`	Vector of registered client agents.
`Type`	`"RegisterNotification"`	Notification of a new client registration.
`Name`	`"CoordinationAgent"`	Address of the coordination agent.
`List`	`ListOfClients`	Vector of registered client agents.
	consumes	
`Type`	`"RegisterClient"`	Registration form for clients.
`Type`	`"DeregisterClient"`	De-registration of a client.
`Type`	`"RegisterCoordAgent"`	Registration form for coordination agents.

5.2.4 Evaluation / Discussion

Private data spaces. We used a solution with a combination of a shared data space for all client agents and a private data space for each observer agent (as shown in figure 5.2). Although this makes the example more complicated, we used private data spaces for two reasons (compare with section 3.8.4). Firstly, private data spaces enhance security by allowing the client agents to do with the forms in their private spaces whatever they like without affecting the rest of the system. We cannot be sure, that every client agent only nondestructively reads a specific form in a global shared data space and we cannot enforce it to do so, but using private data spaces we can be sure, that the system is still working properly even if one client does not follow the rules. Secondly, it allows the clients to work more efficient. In the single data space version the client agents have to poll for new notifications (at least in standard Linda implementations) and to decide whether it has changed or not since the last check they have to save the last notification internally. This kind of a procedure is not at all what we have in mind when we think of a notification. With private data spaces the client agent can be programmed to have a thread blocked, waiting for the desired

notification form to appear in its private data space.

Additional to the private data spaces we used a shared data space for the communication of state requests from the observers and the actual state of the subject. This shared data space offers simplicity, because the client agents can communicate "directly" with each other without the need for a coordination agent to transport the forms back and forth. This design of course has some disadvantages in terms of security as mentioned above, but in this example security is not an issue. By combining the two designs in one example, we showed that the decision for private data spaces or shared data spaces is not exclusive.

De-registration. When an observer agent leaves the configuration, it does not have to explicitly de-register with the coordination medium. This results from our design choice not to allow the explicit deletion of data spaces, because this is not reasonable in open systems (see section 3.8.2 for details). Without explicit de-registration the coordination agents can still put some forms into the observer agents private data spaces even if the agents have left the configuration. Although not necessary in this example, the de-registration of leaving observers (or client agents in general) is desirable in APROCO anyway, because of the automatic garbage collection built-in in the programming language Java. If a client agent that intends to leave a configuration "surrenders" its reference for its private data space by setting all variables that refers to that data space to `null` and de-registers with the coordination medium, the coordination agents inside the medium can themselves surrender all their references to this data space as well and it will eventually be automatically garbage collected. Explicit de-registration of client agents allows us to save unneeded system resources in terms of memory as well as computation, because the coordination agents are not servicing these private data spaces any more. We introduce examples where de-registration of leaving client agents is necessary, because some coordination agents need to know which client agents are actually present.

5.3 Electronic Vote

5.3.1 Problem Description

The example is derived from the one presented in [MU97], which describes a system that enables an open group of agents to hold secret and fair votes on different issues. The rules are as follows:

1. Every member can vote at most once and within a given time period
2. Counting of the votes is done correctly
3. The vote is secret
4. The members are notified about the outcome

Every member can initiate a vote on an issue and there can be several votes going on concurrently, as long as they are on different issues. The concurrent running of votes is an extension to the original setting of the example in [MU97]. Note that the rule (2) is all but trivial to guarantee and thus poses some hard problems to a potential solution. If the counting is done by one of the

participants in the vote, it must be prevented from cheating and manipulating the outcome of the vote.

5.3.2 Coordination Problems

Every client agent (voter) must be enabled to start a new vote. This vote announcement must be multicasted to all the participants. The votes that the participants give must be collected and the rules for the vote must be checked and enforced. The coordination medium is responsible for preventing cheating.

5.3.3 Solution

The solution shown in figure 5.3 consists of a set of voting agents, each with its private data space to put the votes and requests for new votes into, an authentication agent that fetches the votes from the voter agents' private data spaces and assigns an identification to them, a vote controller agent that checks the rules for the vote and runs the timeout for every voting round, the already known multicast and registration agents reused from other examples, and a special global configuration data space with write-only access for the client agents. All the coordination agents share one data space that they use to pass forms containing votes and results around.

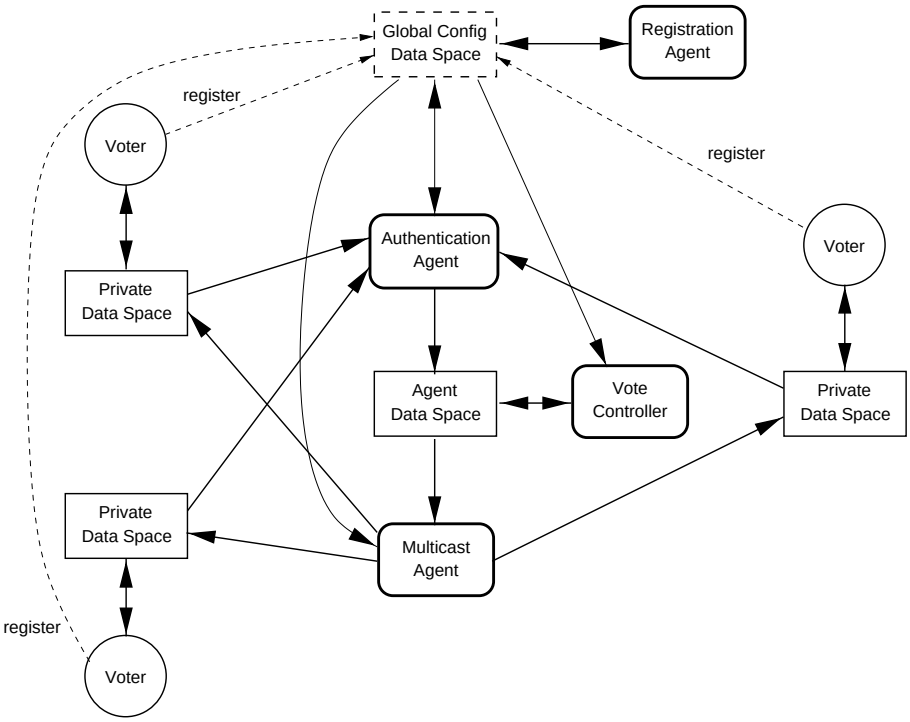


Figure 5.3: Solution for the electronic vote example

Voter. Each voter agent first has to register its presence with the coordination medium through the global configuration data space. This registration includes a reference to the voters private data space to enable the authentication agent to get the forms posted by the voter. If a voter leaves the group it de-registers with the coordination medium. This allows the vote controller to decide whether the issue was accepted or not according to the actual voting policy such as majority or consensus. Every voter agent can initiate a new vote on a specific issue providing the issue and the deadline associated with this vote. The voters have their independent opinion about a issue and vote accordingly. The vote can be either "yes", "no", or "I don't have an opinion about this." The voters are not obliged to give their votes immediately after the announcement of a new voting round, but it should be given within the alloted time slot. Comparing with situations in the real world, it is also imaginable that voters can try to cheat by voting more than once for a specific issue or try to disturb other participants in fulfilling their political duties. It is the the coordination medium's responsibility to prevent them from leaving the path of virtue.
A voter agent has the following interface to the data spaces it is connected to:

Voter Agent: Global Configuration Data Space

creates		
`Type`	`"RegisterClient"`	Registration form.
`Handle`	`ObjectSpace`	Data space reference for private data space.

Voter Agent: Private Data Space

creates		
`Type`	`"Request"`	Request for the coordination medium.
`Subtype`	`"NewVote"`	Initiates a new vote on an issue.
`Issue`	`"the issue"`	The issue of the new vote.
`Deadline`	`1234`	The deadline for the vote round.
consumes		
`Type`	`"Answer"`	Answer from the coordination medium.
`Subtype`	`"VoteOn"`	Give a vote on an issue.
`Type`	`"Answer"`	Answer from the coordination medium.
`Subtype`	`"Result"`	The result of a vote round.

According to an observation by Manuel Günter in his master's thesis [Gün98] an ordinary Swiss voter has the tendency to reject issues that go beyond a specific size (indicated in his solution with an individual value called "intelligence"), this means that those issues are too hard to understand, thus they must be rejected. In our implementation of the example, the voters have an individual but fixed tendency to accept an issue that is influenced by chance. This behavior - although not as realistic - offers more excitement because the outcome of a vote is not as predictable.

Coordination Agents

It is hard to ensure correct counting of the votes if we allow a voting participant that can be personally involved in the outcome of the vote to count them. Furthermore it is much easier to ensure secrecy of the individual votes if there is a "neutral" instance controlling the votes. We split

the coordination problem into smaller parts that can be solved by simpler coordination agents. This allows us to reuse some coordination agents that we already introduced in earlier examples in new settings.

Registration Agent. The registration agent has been presented in section 5.2 and is reused here in exactly the same function.

Authentication Agent. The authentication agent is checking the participating agent's private data spaces for votes and initiations of new votes. To be able to do this, the authentication agent has to check in the global configuration data space whether a new participant has registered and get a reference for its private data space. Unlike e.g. the multicast agent introduced before, the authentication agent does not have a sort of "trigger" form that it could use to update its internal list of registered voters to check their private data spaces. For this purpose it registers its interest in getting a notification each time a new voter has registered with the coordination medium with the registration agent (introduced in section 5.2). This allows the authentication agent to update its list of registered voters exactly when needed. The authentication agent assigns a unique identification to every voter agent and adds this identification to every form it collects from those voters' private data space. This marks the origin of a particular form and enables other coordination agents to check the voting rules such as duplication of votes. The vote forms and requests for new votes are then transported into the shared data space for the coordination agents for further processing.

The authentication agent has the following interface to the data spaces it is connected to. We only list the interface for one private data space, but the interface to all other private data spaces are identical.

Authentication Agent: Global Configuration Data Space

creates		
`Type`	`"RegisterCoordAgent"`	Registration form.
`Name`	`"Authentication"`	Address of this agent.
consumes		
`Type`	`"RegisterNotification"`	Notification of a new client registration.
`Name`	`"Authentication"`	Address of the agent.
`List`	`ListOfClients`	Vector of registered client agents.

Authentication Agent: Agent Data Space

creates		
`Type`	`"Request"`	Request for the coordination medium.
`Subtype`	`"NewVote"`	Initiates a new vote on an issue.
`Issue`	`"the issue"`	The issue of the new vote.
`Deadline`	`1234`	The deadline for the vote round.

Authentication Agent: Private Data Spaces

consumes		
`Type`	`"Request"`	Request for the coordination medium.

Vote Controller. The vote controller processes the votes and the requests for new vote rounds. If a new vote round is requested by a voter agent the vote controller checks if there is already a vote going on for the same issue and if so postpones the new round until the actual round is finished. A new round is started by announcing it to the participants via the multicast agent and initiating the timeout associated with the particular issue. The announcement of a new vote round includes a unique identification of the round to be able to distinguish votes on the same issue but of different rounds. Incoming votes are checked for the rules given above. First they are tested to be for an issue that is currently voted on (if the timeout has expired, the vote is removed from the list of actual vote rounds), this is checked with the identification of the voting round or, if missing, the issue itself. Then it is tested whether this voter agent has already given a vote for this issue in this round, this is checked with the identification for the voter that the authentication agent assigned to every vote. The actual counting of the votes for this issue will then be adjusted according the vote given.

When the timeout has run out the result of the vote round is announced to all the participants through the multicasting agent and the issue is removed from the list of actual vote rounds. To be able to correctly determine whether the issue was accepted or rejected, the vote controller needs to know the amount of voters currently registered with the medium. This information is maintained by the registration agent and can be read out from the global configuration data space. After the announcement of the result the vote controller checks for pending vote rounds for the same issue and starts the first one, if present.

The vote controller has the following interface to the data spaces it is connected to:

Vote Controller: Global Configuration Data Space

reads		
`Type`	`"ListOfClients"`	The list of actually registered clients.

Vote Controller: Agent Data Space

creates		
`Type`	`"Answer"`	Answer from the coordination medium.
`Subtype`	`"VoteOn"`	Give a vote on this issue.
`Issue`	`"the issue"`	The issue of this vote.
`VoteTag`	`5`	An identification for this vote round.
`Type`	`"Answer"`	Answer from the coordination medium.
`Subtype`	`"Result"`	The result of a vote round.
`ProVotes`	`3`	Number of votes for this issue.
`ContraVotes`	`2`	Number of votes against this issue.
`UndecidedVotes`	`5`	Number of votes undecided or too late.
consumes		
`Type`	`"Request"`	Request for the coordination medium.
`Subtype`	`"NewVote"`	Initiates a new vote on an issue.

Multicast Agent. The registration agent has been presented in section 3.7 and is reused here in the same function. It multicasts the announcement of a new vote round on a specific issue and the

results of such a round to all the registered participants. In this example the multicast agent listens to forms with the key/value pair `Type "Answer"` and transports them into the voters' private data spaces.

5.3.4 Evaluation / Discussion

This example shows the enforcement of rules performed by an explicit entity - a coordination agent. The example involves time measurement and requires pro-active behavior of the participants to initiate a new vote. It also shows the advantage of decoupling of agents to prevent cheating.

Cheating. The decoupling of the agents using data spaces is important to prevent cheating in this example, because there are no agent has a reference for another one, thus there is no possibility of an agent to directly disturb another agent. The only remaining possibility to cheat for an agent is to get access to other private data spaces or data spaces that are meant to be exclusively used by the coordination agents. The coordination agents are statically connected to their shared data space in this example, this means that there is no reference to it available outside the coordination agents themselves, thus no client agent is able to get access to this data space. The situation for the private data spaces of the client agents is different. Because there is an open set of participants, new client agents must be allowed to enter the set of participants by joining the actual configuration. For this purpose a client agent creates its own private data space and registers with the coordination medium providing a reference for it. This registration must be done in a shared data space that is well-known in the application, because there must be a starting point for a joining agent to get access to at least one data space to participate in the application. Usually the standard global configuration data space is used for registration purposes, but in this special case where we need to prevent client agents from cheating, the global configuration data space must be secured by restricting the client agents access to it to write-only access. As mentioned in section 3.8.5 only a few implementations of Linda dialects support access restrictions on data spaces. We could not implement real access restrictions, because the Linda implementation we used (called Jada [CR96], see chapter A for details) does not support access restrictions either.

The problem of preventing cheating could only be solved in the combination of the coordination agents, the private data spaces, and the write-only access of the global configuration data space for the client agents.

Centralized Control. The solution we present here uses a centralized instance (the vote controller) to check the rules of the vote. It would be preferable to have a decentralized solution for fault-tolerance reasons. The most preferable solution would be to use one of the voter agents itself (e.g. the one that initiated the new vote) to do the controlling, this is especially useful, if we use this voting for an election of a group leader as used in distributed systems (e.g. to elect a new specialized group member after the former one had gone down). In this case it is extremely difficult to ensure that the controlling voter cannot cheat.

Policies. In this example different policies for the voting are possible. We can use a majority policy as we are used to here in Switzerland or for special issues we could adopt a consensus

policy where all the participants must agree on the issue to have a positive outcome. In both cases the vote controller needs to know the number of participating voter agents[2]. This information can be read out of the global configuration data space where the list of actually registered voters is maintained by the registration agent. To actually change the policy we need to change the example and use another coordination agent that we introduce in the next example, the policy agent.

5.4 Administrator / Worker

5.4.1 Problem Description

This example is derived from a standard architectural pattern for the distribution of tasks in parallel programs. It was first mentioned by Gentleman [Gen81] and cited frequently in other publications, e.g. [Pap95]. A special administrator process distributes requests coming from an arbitrary number of clients to worker processes and collects and returns the results back to the clients. The administrator can employ different policies for the distribution of the client requests. If the worker processes are all identical, the administrator can leave the distribution to the shared data space architecture that our prototype is built upon. In this case, the workers are just looking for new jobs in the shared data space and grab the next one to process it, this corresponds to the *agenda parallelism* model [CG90] of parallel processing and automatically guarantees for optimal load balancing. If in turn the worker processes are different from each other, the administrator must distribute the tasks according to the workers capabilities to satisfy the clients, this corresponds to the *specialist parallelism* model [CG90]. It is even possible that the administrator breaks a single request into smaller subrequests, that can be serviced by several workers in parallel to achieve low response times for the clients and possibly optimal workload balancing for the worker processes.

The policy used for the distribution of requests can be influenced by a variety of factors, like request priorities, size of the requests, minimal response time for requests, optimal workload balancing, etc. As shown in [Tic97] some policies need application dependent information to work properly. For this purpose we use the global configuration data space for the coordination agents to access and possibly change this information.

5.4.2 Coordination Problems

The main coordination problem in this example is to distribute the client requests to the workers transparently following a specific distribution policy.

5.4.3 Solution

The solution shown in figure 5.4 consists of a set of client agents connected to one shared data space, a set of worker agents connected to one shared data space, a workload manager that assigns a target worker agent to each client request and transports it into a shared data space where the fault tolerance agent introduce from section 5.1 can provide its service by starting a timeout and

[2] In case of the consensus voting policy, it is arguable if the number of participants needs to be known. One could have the opinion that a politician that votes too late or is absent should not play a role in such important issues, but the political reality looks somewhat different.

checking for the right answers. There are two coordination agents connected only to the global configuration data space, the worker information agent that collects information about the registered worker agents, and the policy agent that enables the dynamic change of the actually used policy for the distribution of client requests.

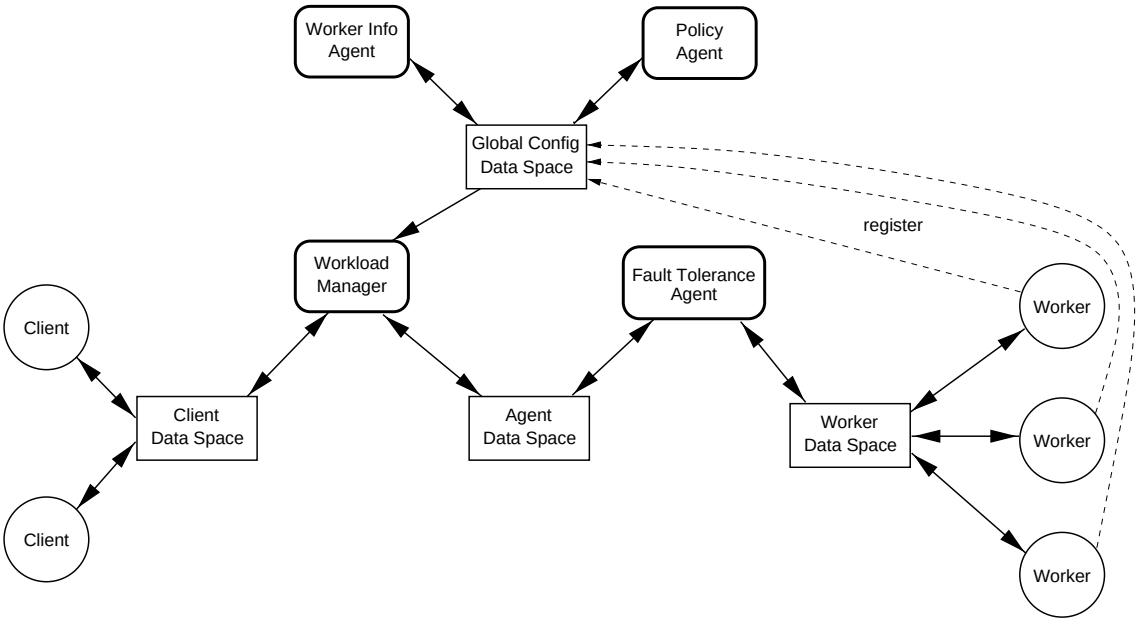


Figure 5.4: Solution for the administrator / worker example

Client. The client agents generate requests for some work to be done and wait for the answers to this requests. Each client agent has a unique client ID that it adds to every request it is posting to the shared data space. The clients listen for answers that are explicitly addressed to them in the shared data space. It is possible that the order of subsequent requests gets changed during their travel through the coordination medium, thus, if the order of the answers is relevant for a specific client agent, it has to take care of this itself, e.g. by simply adding a request number to every request form that will be increased by one every time a new request is generated and posted. The system is designed to return all the information included in the request form, therefore this number will be returned and can be used to restore correct order.

The agents operating as clients need to have the following interface (the ID for the client is an example):

Client: Client Data Space

creates		
`Type`	`"Request"`	Type of the form.
`ClientID`	`5`	The client's ID.
`ComputeTask`	`"Task"`	The task to be computed by a worker.
consumes		
`Type`	`"Answer"`	Type of the form.
`ClientID`	`5`	The client's ID.

Note that the client agents do not have to register and de-register with the coordination medium, because of the design choice to use only one shared data space for all the client agents instead of private data spaces for each client agent.

Worker. A worker agent is able to compute a result for a client request according to its capabilities. It is possible that workers with different capabilities are present in the same application. Therefore they have to register their presence and their capabilities to the coordination medium to enable it to direct the right type of requests to them. To allow the coordination medium to use different distribution policies each worker agent listens to forms either explicitly addressed to it or to all worker agents by addressing them to "anonymous". As discussed later this mechanism is used for an easy exploitation of the built-in capabilities of parallelism in shared data spaces. Each worker returns all the information it got in the request together with the processed result. This serves in the first place to allow the client to grab the answer that belongs to a request originated from this client simply by adding its client ID to the request and getting it returned with the answer. As discussed later, this mechanism prevents the coordination medium from using private data spaces for this purpose. This mechanism also allows new services that introduce new information in the requests and answers to be easily added to the coordination medium. As an example, the fault-tolerance service needs to know which answer belongs to which request, and this can easily be achieved if the worker returns all the information that was included in the request such as a request identification. We will explain this mechanism in the discussion section.

The agents operating as workers need to have the following interface:

Worker: Global Configuration Data Space

creates		
`Type`	`"RegisterWorker"`	Registration form.
`WorkerID`	`17`	The worker's ID.
`Capabilities`	`CapabilityObject`	Description of the workers capabilities.

Note that the worker agents must register and de-register with the coordination medium, although they share one data space in the same way as the client agents. The registration and de-registration of the worker agents is needed for the workload manager to work properly.

Worker: Worker Data Space

creates		
`Type`	`"Answer"`	Type of the form.
`ClientID`	`5`	The client's ID.
`WorkerID`	`17`	The worker's ID.
`ComputeTask`	`"Task"`	The task to be computed by a worker.
`Result`	`"Result"`	The result for the task.
consumes		
`Type`	`"Request"`	Type of the form.
`WorkerID`	`17`	The worker's ID.
`Type`	`"Request"`	Type of the form.
`WorkerID`	`"Anonymous"`	No special ID required.

Each worker agent has to listen for two "addresses" of the requests: either their own ID (in the example 17), or "anonymous". This is used to exploit the built-in parallelism in case of identical worker agents.

Coordination Agents

Worker Information Agent. The worker information agent is a variant of the registration agent introduced in section 5.2. The worker information agent is also looking for new client agents (in this case the worker agents) that are registering with the coordination medium and maintains a list of some of their properties. The worker information agent is only maintaining a form in the global configuration data space containing the identifications of the workers and their processing capabilities. There is no registration of other coordination agents or notification of coordination agents as with the registration agent.
The worker information agent has the following interface to the global configuration data space:

Worker Information Agent: Global Configuration Data Space

creates		
`Type`	`"ListOfWorkers"`	The list of actually registered workers.
`List`	`WorkersAndCaps`	Vector of workers and their capabilities.
consumes		
`Type`	`"RegisterWorker"`	Registration form.
`Type`	`"DeregisterWorker"`	De-registration of a worker.

Fault Tolerance Agent. The fault tolerance agent has been presented in section 5.1 and is reused here in exactly the same function.

Policy Agent. The policy agent enables dynamic change of the actually used policy for an action, in this case the distribution of requests among an open group of worker agents. The actually used policy is encapsulated into a *policy object* that the policy agent puts into the global configuration data space. We used object-oriented design practice that is available for our implementation

language Java to realize this policy object. We employed the Strategy pattern [GHJV95] to be able to dynamically exchange the actually used policy.

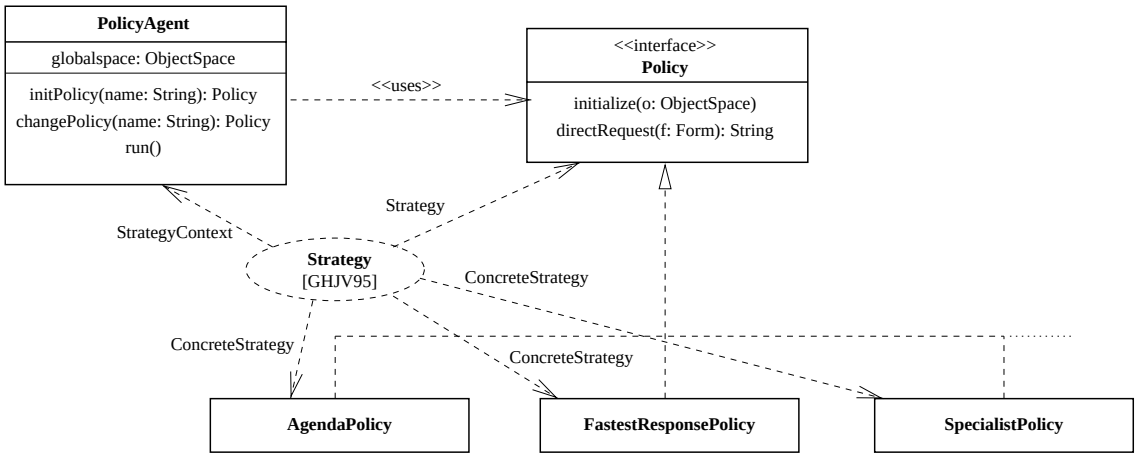


Figure 5.5: Design of the policy interface

Figure 5.5 shows the design that we used to realize our policy object. We defined a *Policy interface* that each concrete policy object has to implement with two methods: `initialize` to allow the policy to read in configuration information gathered in the global information data space and `directRequest` to assign a target worker agent to this specific request. The `initialize` method is used for application dependent information that is needed when the policy has changed and has to be set up to be operational. Application dependent information in this example is e.g. the individual capabilities of a registered worker agent that has to be read in during run- time.

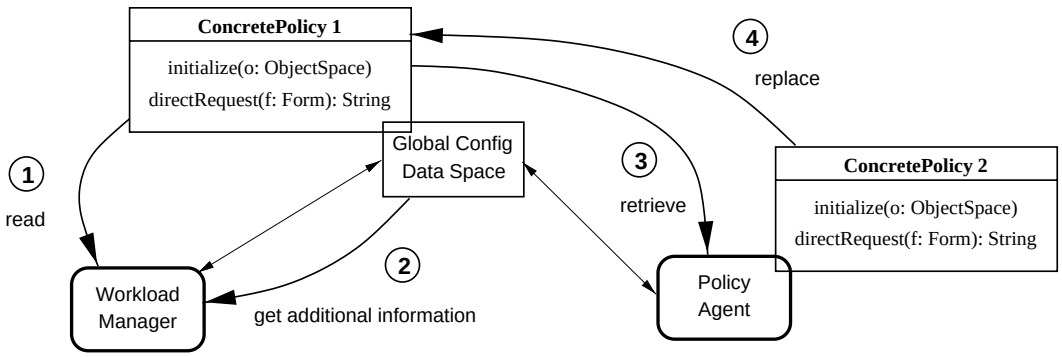


Figure 5.6: Dynamic change of used policy

The dynamic change of the actually used policy is shown in figure 5.6. Every time a new request arrives the workload manager reads in the actual policy object (1) and any additional application dependent information also stored in the global configuration data space (2). When the policy agent wants to change the actual policy it simply replaces the actual policy object (3) in the global configuration data space by a new one implementing the same interfaces (4). This

causes the workload manager to use the new policy object when it assigns a worker ID to the next client request.

The policy agent has the following interface to the global configuration data space:

Policy Agent: Global Configuration Data Space

creates		
`Type`	`"Policy"`	A policy for the workload manager.
`PolicyObject`	`PolicyObject`	An object encapsulating the policy.
consumes		
`Type`	`"Policy"`	A policy for the workload manager.

Workload Manager. The workload manager looks for requests in the shared data space of the client agents and assigns a worker identification as address to each request according to the actual policy. This policy is encapsulated into a policy object that the workload manager reads out from the global configuration data space every time a worker ID has to be assigned to a request. This identifications of the workers are collected by the worker information agent and can be read out from the global configuration data space. The used policies can be dependent on additional information about the workers such as their capabilities. This information can also be read out from the global configuration data space. Because of the openness of the group of workers the IDs as well as additional information must be read out each time a target worker ID gets assigned to a request.

The workload manager has the following interface to the data spaces it is connected to:

Workload Manager: Global Configuration Data Space

reads		
`Type`	`"ListOfWorkers"`	The list of actually registered workers.
`Type`	`"Policy"`	A policy for the workload manager.

Workload Manager: Client Data Space

creates		
`Type`	`"Answer"`	Type of the form.
`ClientID`	`5`	The client's ID.
`WorkerID`	`17`	The worker's ID.
`ComputeTask`	`"Task"`	The task to be computed by a worker.
`Result`	`"Result"`	The result for the task.
consumes		
`Type`	`"Request"`	Type of the form.

Workload Manager: Agent Data Space

creates		
`Type`	`"Request"`	Type of the form.
`ClientID`	`5`	The client's ID.
`ComputeTask`	`"Task"`	The task to be computed by a worker.
consumes		
`Type`	`"Answer"`	Type of the form.

5.4.4 Evaluation / Discussion

Dynamic Change of Policy. In this example we showed an extension to the idea of *pluggable policies* as presented in [Tic97]. We showed a solution for dynamic policy change using shared data spaces. To achieve this, we had to exploit object-oriented design practice and encapsulate the policy behavior into a policy object that is consulted each time a new request arrived. The dynamic change of the policy is performed by simply exchanging this policy object in the global configuration data space. This causes the workload manager to use the new policy the next time it consults the policy object from the global configuration data space.

Exploit Built-In Parallelism. Instead of using a private data space for each worker agent where it is hard to realize load-balancing, we make use of the built-in capabilities for parallelism in shared data spaces. When several identical agents are connected to one shared data space we automatically have agenda parallelism. Every agent takes a new request as soon as it is ready to do so, there is a guarantee for optimal load- balancing among the agents. To be able to exploit this feature but still be able to use explicit addressing for the situation that we have worker agents with different capabilities (for a specialist parallelism policy), we let the worker agents listen for explicitly addressed requests and for those that are meant to be taken by the first one that is free. This requests are simply addressed "anonymous" and are used to simulate the described situation of non-addressed requests.

5.4.5 Possible Improvements

If repeated requests are not answered the fault tolerance agent could signal the suspected breakdown of a worker agent to the other coordination agents (via the global configuration data space). The worker information agent could then remove this worker agent from its list of registered workers to prevent the workload manager to assign new client requests to it. The breakdown of a worker agent could lead to problems especially in specialist parallelism situations where there is possibly no other worker with the same working capabilities. In this situation a request has to be refused if there is no worker agent able to process it.

5.5 Discussion of the Examples

We realized this coordination examples with APROCO to show its ability to express flexible and reusable coordination abstractions in a straight- forward way.

Table 5.1 shows where the different coordination agents that we introduced in chapter 3 are used in the sample applications. The symbol $\surd$ means that the indicated coordination agent was used in the sample application.

Coordination Agent	Sample Applications					Reference
	Multicast	Fault Tolerance	Observer	Electronic Vote	Administrator / Worker	
Registration			$\surd$	$\surd$		Section 5.2
Client Information					$\surd$	Section 5.4
Policy					$\surd$	Section 5.4
Order						Section 3.8.2
Transport						Section 3.8.2
Fault Tolerance		$\surd$			$\surd$	Section 5.1
Vote Controller				$\surd$		Section 5.3
Multicast	$\surd$		$\surd$	$\surd$		Section 3.7
Collector						Section 3.5
Authentication				$\surd$		Section 5.3
Workload Manager					$\surd$	Section 5.4

Table 5.1: Overview of the used coordination agents in the sample applications

The transport agent and the order agent were not used in this set of sample applications. We introduced both coordination agents in chapter 3 where we explained the dynamic re-composition of coordination agents. The collector agent is a simplified version of the authentication agent. It collects forms from private data spaces, but without assigning an ID to them. We did not use it in any of the sample applications, but considered it nevertheless useful for designs using private data spaces.

Reusability. We can see from the table 5.1 that there are several coordination agents that have been used in different examples. This shows the easy reusability of this kind of coordination abstractions.

Flexibility. The dynamic exchange of policies shown in the Administrator / worker example in section 5.4 is an example of the run-time flexibility of the used coordination abstractions.

5.5.1 Classification of the Used Coordination Agents

The coordination agents that we presented in section 3.5 and used in the examples can be classified as follows:

- Communication / Synchronization
  - Transport agent
  - Multicast agent
  - Collector agent
- Management of the coordination medium
  - Policy agent
  - Registration agent
  - Client information agent
  - Order agent
- Enforcement of constraints
  - Vote controller
- Special services
  - Fault tolerance agent
  - Authentication agent
  - Workload manager

Due to the use of generative communication and shared data spaces the communication purposes of a coordination agent cannot be separated from its synchronization purposes.

Chapter 6

Conclusions

With APROCO we introduced a programmable coordination medium for agent-based open systems in this thesis. APROCO supports the clean separation of concerns by allowing to separate the coordination aspects of an application into explicit entities, the coordination agents. These explicit entities use Linda-like primitive operations on shared data spaces to achieve the coordination of the client agents' interactions. Standard Linda offers a high degree of decoupling of agents which is a necessary property in open systems, but it does not offer the clean separation of coordination aspects from the rest of the application in terms of reusable entities. Furthermore, Linda only offers a small set of "add on" primitives that are easy to understand and embed into a host programming language, but only offers very limited functionality in terms of realistic coordination abstractions.

We first present a summary of the problems that lead to the proposition of a coordination medium, then we describe the main parts of our coordination medium APROCO and list the results we found during the investigation. We conclude the chapter with the outlook for further work in this domain.

6.1 A Programmable Coordination Medium

Problem Description

To keep up with rapidly changing requirements applications are increasingly built out of software components. This allows one to adapt or replace only the affected component instead of the whole application when the requirements change. A new trend is now to give those software components control over their own actions, to turn them into concurrently running software agents. These software agents have to be relatively independent to keep them exchangeable. Although quite independent, they still need to interact in order to achieve the application's overall goal. This results in the need to coordinate their interactions. When requirements change, the used coordination solution may need to be changed as well. This results in the need to express the coordination solution in a flexible way.

A number of coordination models were created to express common coordination solutions. Linda is one of the most prominent representatives of such coordination models. Linda is widely

used because it offers means to separate coordination code from computational code within an agent and a high degree of decoupling of agents to increase their independence. However, Linda does not strictly separate coordination aspects from computational aspects in a whole application. Furthermore, Linda only offers a set of primitive operations and leaves the user with the task to construct realistic coordination abstractions out of them.

Coordination abstractions are often hard-coded into the participant agents' protocols and therefore neither flexible nor reusable. They are spread all over the application and it is almost impossible to identify them (section 2.2.2).. It would ease application as well as reuse of the coordination solution, if we could identify and encapsulate those abstractions explicitly. It is not easy to encapsulate coordination abstractions because coordination typically affects multiple agents, and in open systems other requirements, such as flexibility and security, must also be dealt with.

This results in the need for a coordination environment in which coordination solutions can be expressed as explicit entities to make them both flexible and reusable.

APROCO

We propose that to be able to reuse a coordination solution it is necessary to abstract it from the coordinated agents and to encapsulate it into an explicit entity, a *coordination agent*. We want to use agents to encapsulate the coordination abstractions, because agents allow us to describe coordination services that are not merely reactive, they can behave pro-active (section 3.4.3).

We propose that by combining coordination agents into a *coordination medium* that delivers all the desired coordination services to its client agents, it is possible to take the coordination aspects out of the client agents' protocols and make them cleaner and easier to understand and reuse as well. A coordination medium is the means to enable communication between the agents and serves to aggregate agents into an ensemble (see section 2.2.1). Our coordination medium APROCO that we describe in this thesis is more than a coordination medium as just defined, it offers coordination services to its client agents that can be exchanged and modified and is thus a *programmable coordination medium* as coined by Denti et al. [DNO97] (see section 2.3.3)..

In this thesis we present the architecture of our coordination medium APROCO. APROCO is based on generative communication and uses the standard Linda operations to allow the client agents to communicate with each other while being as independent from each other as possible (section 3.4.1). The information exchanged between the client agents is wrapped into forms (section 3.8.6) and stored in shared data spaces. We used Linda operations as the basic coordination level because of their simplicity and easy implementation in usual programming languages. To overcome the limitations of Linda we encapsulated the coordination solutions into explicit entities, the coordination agents (section 3.4.3). They are coordinating the activities of the client agents connected to the coordination medium using the same basic Linda operations on shared data spaces to inspect and possibly transform the flow of information between the client agents.

Important Results of APROCO

List of Useful Coordination Abstractions. We present a list of useful coordination abstractions encapsulated into coordination agents that can be used within APROCO. We took an experimental approach to find these coordination abstractions by investigating small but typical real-world

applications. The presented list of coordination agents is not complete and can be extended as needed. It can be found in section 3.5.

Uniform Communication Model for all Agents. A coordination agent in APROCO does not have any special means to get involved with the communication between client agents than through inspection of the data spaces that it has access to. This implies that - without leaving the generative communication style - it is not possible to give the coordination agent a higher priority to read or consume a particular form in a data space than a client agent competing for the same form. In some cases the client agent would be able to consume the form before the coordination agent could read it or perform some transformations on it. It is thus necessary to use more than one single shared data space - as with traditional Linda - for all the agents of an application to allow the coordination agents to fulfill their coordination purpose.

Need for Multiple Data Spaces. The uniform communication model described in the preceding paragraph is a direct cause for the need for multiple data spaces (section 3.8.3). But even if we would introduce special means for the coordination agents to get priority over the client agents in getting the forms in the shared data spaces, the need to be able to split an applications into subconfigurations for the easier construction of large systems results in the need for multiple data spaces, because it is not possible to create subconfigurations in a useful way using only one single shared data space (section 3.8.3).

Need for Private Data Spaces. We introduced the notion of a private data space for a data space that is only accessible for one client agent (typically the client agent that created this data space) and the coordination medium, but not for other client agents. We found that designs using private data spaces have a lot of advantages over designs using shared data spaces for several client agents. Private data spaces are important for security reasons, liveness considerations, and if a client agent needs exclusive access to some data (section 3.8.4).

Access Rights On Data Spaces. To enable security, it is vital to have the possibility to define different access rights on shared data spaces. Because APROCO supports multiple data spaces, it is sufficient to be able to define access rights per data space and, because of its agent architecture, per agent (section 3.8.5).

Important Properties of APROCO

Separation of Concerns. APROCO supports the clean separation of concerns. The client agents are mainly concerned with the computational aspect of the application - besides the communication with other client agents necessary to reach the overall goal of the application - and the coordination agents are solely concerned with the coordination aspects of the application.

Coordination Abstractions as Explicit Entities. APROCO factors out the coordination solutions into explicit, dynamically changeable entities, the coordination agents. This increases or

even enables reusability and flexibility of the used coordination abstractions. The coordination agents can be combined to deliver high-level coordination services to the client agents.

Flexibility and Reusability Through the explicit representation of coordination abstractions as coordination agents we gain flexibility and reusability of this abstractions.

Open Set of Coordination Agents. The list of coordination agents that we presented in section 3.5 is only meant to give an idea of how coordination abstractions can be implemented in APROCO. The set of coordination agents is open and can be extended as new coordination abstractions are found. Existing coordination agents can be used as skeleton to build similar coordination abstractions.

Decoupling of Agents. The use of generative communication as the basic means for all communication among the agents delivers a high degree of decoupling of producers and potential consumers of data.

Dynamic Re-composition. The loose coupling of the agents enable them to dynamically join or leave a configuration without leaving others with broken communication channels. Together with the possibility for the agents to create new data spaces and attach to existing data spaces this yields the means for dynamic composition and re-composition of applications as described in section 3.8.2. To be able to tackle the changing requirements in open systems, dynamic re-composition must be supported by any serious candidate architecture in this domain.

Extensibility. The functionality of the client agents can easily be extended through the introduction of new labels in the forms that are used to wrap the information. These new tags do not interfere with the functionality of other agents, they are simply ignored by those agents that do not know them.

6.2 Dynamic Exchange of Policies

We incorporated an extension to the idea of *pluggable policies* as presented in [Tic97] into our coordination medium APROCO. If an action can be performed according to different policies, it is a desirable property to be able to exchange the used policy to be able to adapt the application to changed requirements without changing other parts of the application as well. In [Tic97] the chosen policy could be "plugged" into the application at system startup using a parameter for the application. In APROCO we incorporated the possibility to dynamically exchange the actually used policy (see the administrator / worker example in section 5.4 for an application).

The actually used policy is encapsulated into a *policy object* that a coordination agent puts into the global configuration data space. We used object-oriented design practice that is available for our implementation language Java to realize this policy object. We employed the Strategy pattern [GHJV95] to be able to define objects that can be dynamically exchanged without breaking the behavior of the application. We defined a *Policy interface* that each concrete policy object

has to implement with two methods: `initialize` to allow the policy to read in configuration information gathered in the global information data space and `directRequest` to assign a target worker agent to this specific request. This method is application dependent and may be defined different according to the application domain. The `initialize` method is used to get application dependent information that is needed when the policy has changed and has to be set up to be operational. Such application dependent information can be special capabilities of the client agents that need to be payed attention to in the policy.

To illustrate this mechanism, we repeat a figure that was used in the administrator / worker example in section 5.4.

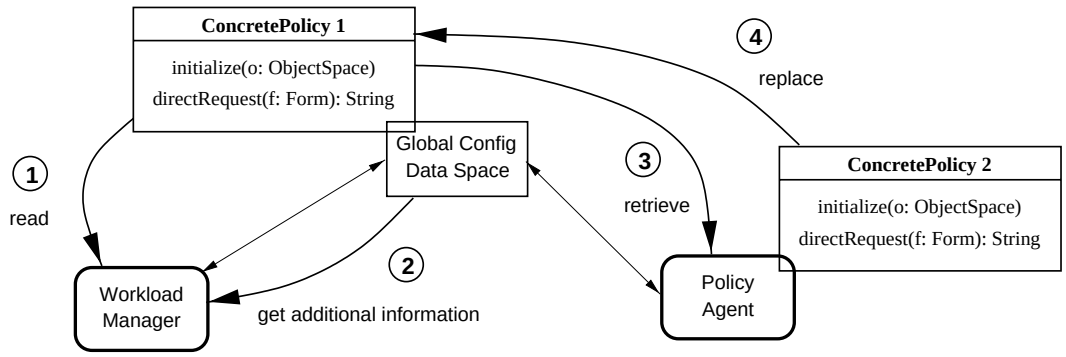


Figure 6.1: Dynamic exchange of actually used policy

Every time a coordination agent is about to perform an action according to a specific policy it reads out the corresponding policy object from a policy form stored in the global configuration data space (1). This policy object is then used to actually perform the action. If additional information is needed the policy object can fetch this information from the global configuration data space as well (2).

The dynamic exchange of the actually used policy is shown in figure 6.1. A special coordination agent, called *policy agent*, simply removes the policy form containing the former policy object from the global configuration data space (3) and replaces it with a policy form containing the new policy object (4). The next time the coordination agent - in figure 6.1 called workload manager - is about to perform the policy guided action, it reads out the policy form in the global configuration data space containing the changed policy and thus the action will be performed according to the new policy.

6.3 Future Work

This project has been, as far as we know, the first attempt to combine the decoupling of shared data spaces with the capabilities of agents to encapsulate the coordination abstractions. As a consequence, the results presented in this thesis are first results and further research work needs to be done to generalize the model.

Distributed Environment. We implemented APROCO in a non-distributed environment using Java threads as implementation of the agents that run within the same Java virtual machine. This way there was no need to pay attention to the problems of real distribution such as network failure, network latency, etc. Furthermore, we did not have to bother with distributed object references that would have been a problem with dynamic creation of data spaces. We believe that the best way to tackle these problems is to use a distributed systems infrastructure that already offers shared data spaces such as Sun's JavaSpaces [Sun98] and implement APROCO on top of this infrastructure. It has to be investigated whether the decoupling of shared data spaces are sufficient to overcome the additional problems posed by real distribution of an application.

Non-centralized Coordination Solutions. The coordination solutions we present in this thesis use centralized coordination solutions. We often use one single coordination agent to perform an important coordination service. This is not desirable in distributed environments because of fault tolerance considerations. We need to investigate whether our concepts can be used to construct real distributed coordination solutions.

Scopes. The notion of forms and the corresponding matching operation we present in this thesis does not support scopes. If a value of a key is overwritten, there is no way to retrieve it later on. The notion of forms could be extended to support scopes, in the sense that new key/value bindings would just be added to the existent ones and be consulted according the actually valid scope.

Transactions, Events and Databases. Sun and IBM are working on distributed Java-based shared data space implementations to support users with ubiquitous networking, persistent data, transactions, queues, and shared computer performance. Sun is working on its JavaSpaces [Sun98] architecture for distributed computing and IBM on its T Spaces [IBM98a] research project to bring together most recent database technology with the concept of shared data spaces. None of the two systems directly supports explicit abstractions for coordination, but with these systems as underlying architecture, it would be very promising to study coordination problems using APROCO's coordination agents.

Meet the Formal Approach. The research group is working on a formal approach to agents communicating with forms [LAN98] based on process calculi such as the π-calculus. Their approach comes bottom-up while the approach described in this thesis is a top-down approach starting with application requirements from real world applications. In a further effort these two approaches need to be combined to give one the necessary groundwork and the other one the desired connection to real world application requirements. The goal of this connection is to be able to proof some properties of the agents independently from their context.

Appendix A

Jada

Jada ("Java Linda") [CR96] is a combination of Java with the coordination model Linda presented in section 2.2.3. Jada is a Java class library designed for the use in distributed applications for the World-Wide-Web (WWW). Jada provides:

- multiple shared data spaces: shared data spaces can be created as usual Java objects and provide methods for the usual basic Linda operations to be executed on them. Like in standard Linda there are blocking and non-blocking operations available.
- multi-threading support: different Java threads can access the same shared data space; blocking requests are managed at thread-level.
- dynamic creation of data space items: being a normal Java object, a data space item can be created using `new`. The standard data space item in Jada is an instance of the class `Tuple`. This object can act as a container for other Java objects for the fields of the data space item.
- distributed coordination support: shared data spaces can be set up as a data space server for the access over a TCP/IP connection using Java sockets internally. Possible network errors are handled through special Java exceptions that can be caught and handled by the client application or applet.
- mobile object coordination: Jada is not a syntax extension for Java, just a set of Java classes. The Java byte code compiled out of source code using Jada classes can be run on any Java virtual machine. Javas capabilities of dynamically loading and instantiating classes at run-time can be used to implement mobile agents using Jada for the coordination.

Jada is freely available for research projects and can be downloaded from its creators home page (URL: `http://www.cs.unibo.it/~rossi/jada/index.html`).

In the following sections we give an overview of the functionality and the methods that Jada offers, as well as some details about its object model and the pattern matching operation it uses. We conclude this chapter with some remarks about our implementation of forms using Jada.

A.1 Object Spaces

The shared data spaces in Jada are called *Object Spaces*, because the data space items they contain are Java objects instead of the tuples in standard Linda. Only Java objects that implement a Java interface called *JadaItem* can be stored into and retrieved from a Jada Object Space. This interface defines methods for dumping the object into a data stream, restoring the object from a data stream, and a matching operator to compare objects. Jadas Object Spaces provide the following Linda-derived operations on them:

- `out(Object)`: puts a Java object that implements the *JadaItem* interface into the Object Space. The calling thread continues immediately.
- `in(Object):Object`: compares the objects in the Object Space with the supplied template object and returns a "matching" object. The comparison is done by type and using the matching operator defined for the specific object (see next section for details). If a matching object is available in the Object Space, it is removed from the Object Space and delivered back as result of the operation. If no matching object is available in the Object Space, the calling thread is blocked until there is one available. If more than one matching object is available, one is chosen arbitrarily.
- `read(Object):Object`: has the same behavior as `in`, except that a matched object remains in the Object Space.
- `in_nb(Object):Object`: has the same behavior as `in`, except that the calling thread continues immediately. If no matching object is available in the Object Space, the operation returns "null" as result.
- `read_nb(Object):Object`: has the same behavior as `in_nb`, except that a matched object remains in the Object Space.

There is no `eval` operation available in Jada. In Linda this operation is used to create a so called live tuple that has fields that needs to be processed before this tuple finally ends up as normal passive tuple in the tuple space. Using Jada, this behavior can be simulated by explicitly starting new Java threads that compute a result and put this result in the Object Space.

A.2 Object Matching in Jada

As the `in` and `rd` operations of standard Linda are based on tuple matching (described in section 3.8.6), the corresponding `in` and `read` operations of Jada are based on *object matching*. Every object that can be stored in an Object Space in Jada has to define a method inherited from the `JadaItem` interface with an appropriate matching operator for this type of object. Object matching is done in a object-oriented way, i.e. an instance of a subclass of a specific object's class can match this particular object using the same matching operator. With composite objects the matching is done recursively.

A.2.1 The Tuple Class

The standard object container in Jada is the class *Tuple*. Tuple represents the behavior of tuples in Linda, but enhanced with the possibility to store Java objects in the tuple fields. A Jada tuple is a ordered set of items. Each item of a Jada tuple can be either an Integer, a String, a class that implements the interface `JadaItem` or a Class object. A Class object represents a kind of "joker" that matches all objects of this class or a subclass of it. Tuples are used for the Object Space operations in an associative way using a matching operation. Two Jada tuples are matching if they have the same number of items and the i^{th} field of the first tuple matches the i^{th} field of the second one for each i. Two tuple items are matching if:

- they are Integers and they represent the same value.
- they are String and they have the same contents.
- they implement the `JadaItem` interface and the `matchItem` method returns true.
- they are both Class objects representing a class and a subclass of this class.
- one is a Class object and the other is an instance of that class or of a subclass of this class.

```
Tuple ta = new Tuple(new Integer(3), "three")
Tuple tb = new Tuple(new Integer(3), String.class)
```

Figure A.1: Tuple matching in Jada: tuple tb provided as template matches tuple ta.

Figure A.1 shows an example of the matching defined with the Tuple class in Jada. The construct `String.class` is Java 1.1 syntax and returns a class object for the `java.lang.String` class. In Java 1.0 the correct syntax for this was `new String().getClass()`.

The Tuple class offers different constructors according to the number of items the new tuple should have. This means that the arity of a tuple is fixed in Jada. Tuple implements the `JadaItem` interface itself, thus a tuple can contain another tuple as a valid item. Using this mechanism we can construct tuples of arbitrary arity by tuple nesting.

A.3 Remote Access to Object Spaces

Jada offers some classes that enables the construction of Object Space servers that can be remotely accessed by Object Space clients using an IP network. An Object Space server is created using the `jada.net.ObjectServer` class. This class can also run as a stand-alone Java application and is then listening on the given port for requests for the Object Space that it handles. An Object Space server is multi-threaded and runs an own thread for every client request. Those threads are synchronized like other threads that are using a local Object Space for coordination.

Figure A.2 shows a situation with two Object Space servers and a number of Object Space clients that are coordinating using these remote Object Spaces. Object Space clients can be stand-alone Java applications or Java applets.

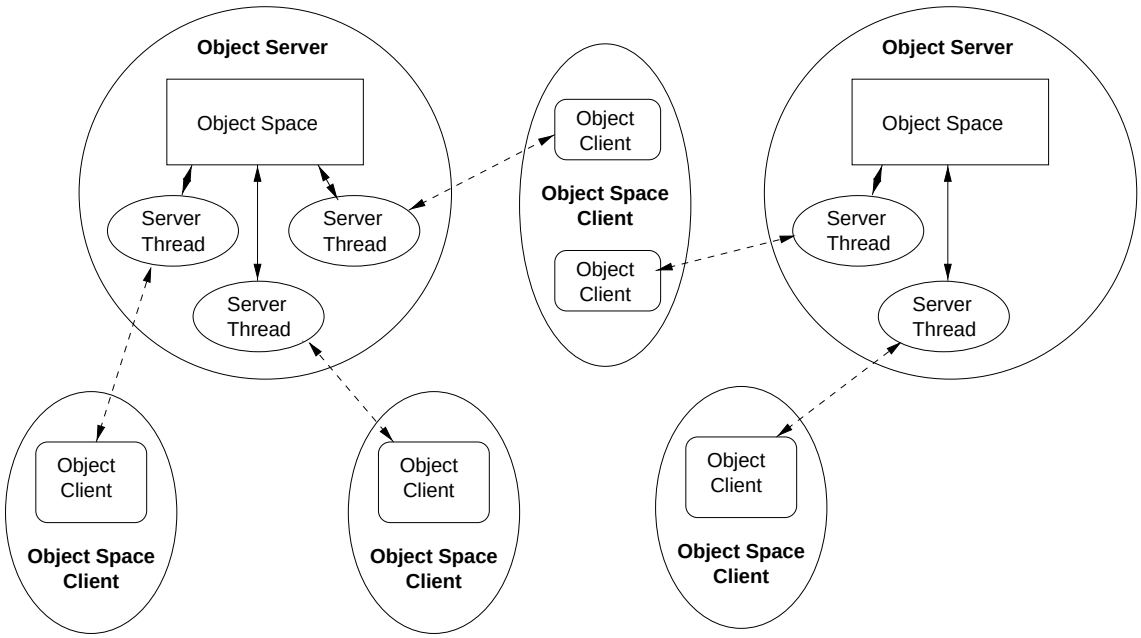


Figure A.2: Coordination of remote applications using the Jada ObjectServer class

A.3.1 Example: Remote Ping-Pong

The following listings in figure A.3 and A.4 show a simple example of the remote coordination provided by Jada. It is an example of two different Object Space clients - one stand-alone Java application and one Java applet. The Object Space server needed for the example application does not need any additional code to the one supplied by Jada itself. One only needs to start an Object Space server on a host with `java jada.net.ObjectServer`. This sets up an Object Space server listening to socket connections on a predefined standard port address. One can also choose an own port address and start the server with this parameter.

Because of the Java "sandbox" security mechanism in Java 1.1 an applet as shown in figure A.3 can only build up a connection to the host it was downloaded from. This means, that the Object Space server must be running on the same host as the Web server to allow this applet to work properly. In Java 1.2 this security mechanism will be changed to allow more flexible security policies.

A.4 Limitations

No access rights on data spaces. Jada offers no means for restricting access to a data space or the items in a data space. As we discussed in section 3.8.5 access rights on data spaces are essential for a coordination medium to be able to offer security for the client agents.

Fairness. Jada is multi-threaded and offers blocking of threads or remote applications over its Object Spaces on the level of single threads. However, Jada uses the thread management capabili-

```
import jada.*;
import jada.net.*;                                  // Classes for remote Jada data spaces.
import java.applet.Applet;
import java.awt.*;

public class PingApplet extends Applet implements Runnable {
  ObjectSpace object_space = null;
  List list = null;

  public void init() {
    object_space = new ObjectClient(getCodeBase().getHost());
                        // Connect to remote data space running on the web server machine.
    setLayout(new BorderLayout());
    list = new List(10, false);
    add("Center", list);
  }

  public void start() {
    new Thread(this).start();
  }

  public void run() {
    while(true) {
      Tuple in = (Tuple)object_space.in(new Tuple("ping",
                                        Integer.class));
      int cnt = ((Integer)in.getItem(1)).intValue();
      object_space.out(new Tuple("pong", new Integer(cnt+1)));
      list.addItem(in.toString());
      list.makeVisible(list.countItems()-1);
    }
  }
}
```

Figure A.3: The Ping Applet: an Object Space client using Jada

```
import jada.*;
import jada.net.*;                                  // Classes for remote Jada data spaces.

public class Pong implements Runnable {

  ObjectSpace object_space = null;

  public Pong(ObjectSpace ts) {
    object_space = ts;
  }

  public void run() {
    while(true) {
      Tuple in = (Tuple)object_space.in(new Tuple("pong",
                                        Integer.class));
      int cnt = ((Integer)in.getItem(1)).intValue();
      object_space.out(new Tuple("ping", new Integer(cnt+1)));
      System.out.println(in);
    }
  }

  public static void main(String args[]) {
    ObjectClient object_space=null;
    if(args.length==1) {                            // Host with remote data space as argument.
      object_space=new ObjectClient(args[0]);       // Remote data space on given host.
    } else {
      object_space=new ObjectClient();              // ''Remote'' data space on this host.
    }
    object_space.out(new Tuple("ping", new Integer(0)));
    new Pong(object_space).run();
  }
}
```

Figure A.4: The Pong Application: an Object Space client using Jada

ties of standard Java, and they are quite poor. Because Java is designed to be platform independent, the requirements for the Java virtual machine have to be in a way that they can be met on every platform. For the scheduling of threads this means, that fairness even in the weakest sense cannot be guaranteed [Lea97]. Running threads are only preempted by threads running with a higher priority. In our experiments we found out that one has to explicitly yield the processor to another thread to give it a chance to grab a shared resource in a Jada Object Space. This can be done by explicitly calling `Thread.yield()` after the shared resource has been released and before another try to get hold on it is started.

A.5 Form Implementation in Jada

In Jada new data space items can be defined with their own pattern-matching behavior by implementing the `JadaItem` interface. We defined an implementation of our forms as presented in section 3.8.6 this way. The resulting `Form` class is shown in figure A.6. For compatibility reasons we used our Form objects wrapped into the standard Tuple objects used by Jada. A form is then represented by a Form object as the only item in a Jada Tuple object.

```
(1) Tuple t = (Tuple)os.in(new Tuple(form));
(2) Form  f = (Form)t.getItem(0);
```

Figure A.5: Access of a Form object wrapped into a Tuple object

To access a Form object in a shared data space one has to follow a special idiom shown in figure A.5. First a Tuple object is searched in the shared data space by providing a pattern Form object wrapped into a Tuple object using `new Tuple(form)` (1). The resulting object is a Tuple object. The first item of this Tuple object is the wrapped Form object. It can be accessed with the `getItem(Position: int)` method of the Tuple object (2).

For the implementation of the forms we used the Java class library JGL [Obj97] from ObjectSpace Inc. in the version 3.1. This library offers a bunch of highly customizable object containers and algorithms for Java and is freely available with some restrictions for commercial use at ObjectSpace's home page (URL: `http://www.objectspace.com/jgl/`). Our Form class is based on JGL's `HashMap` class and uses its methods to add, access, or remove key/value pairs.

A.6 Special Operations on Forms

For convenience we added two special operations on our form implementation, namely *merge* and *update*. Both operations compare two forms and complete one with the missing key/value pairs from the other form.

Figure A.7 shows the principle of the two operations. They are almost specular: `A.merge(B)` and `B.update(A)` yield the same resulting form, but `update` additionally updates the form that it is called on to the resulting form as a side-effect. Both operations are used in situations where one needs to return the unchanged part of a form in a result form to prevent any other interested

```
package aproco.lib;

import jada.*;
import com.objectspace.jgl.*;                    // The JGL class library.
import com.objectspace.jgl.predicates.*;

public class Form extends HashMap implements JadaItem {

  /** Constructs an empty form. */
  public Form() {super(); }

  /** Constructs an empty form with a JGL binary predicate used
    * for comparison of two keys. */
  public Form(BinaryPredicate comparison) { super(comparison); }

  /** Constructs a form as a copy of another form. */
  public Form(Form form) { super(form); }

  /** Tests if two forms match.
   * This method will be invoked on every item in the object
   * space to check if it matches with the template item
   * (passed as parameter), so the object space item is 'this'
   * and the template form is called 'form' here.
   *
   * The test form (form) matches a object space form, if
   * a) the form's keys are a subset of the tuple space tuple and
   * b) the values of this keys are equal */
  public boolean matchesItem(JadaItem item) {
    boolean match = true;
    Form form = (Form)item;
    Object thisvalue;
    if (form.size() <= this.size()) {
      for (HashMapIterator i = form.begin();
           !i.equals(form.end()); i.advance()) {
        thisvalue = (Object)this.get(i.key());
        if (thisvalue == null) { match = false; break; }
        else if (!thisvalue.equals(i.value())) {
          match = false; break;
        }
      }
    }
    else match = false;
    return match;
  }
}
```

Figure A.6: The Form class: implementation of forms using Jada and JGL

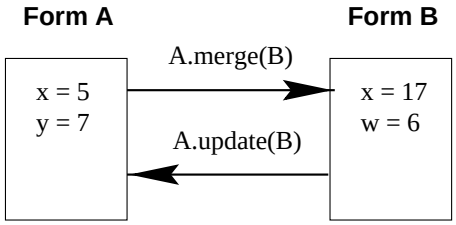


Figure A.7: Special operations on forms: `update` and `merge`

agents from the need to save these parts locally. We use these operations in APROCO to offer easy means for extensibility as described in section 3.8.6.

A.6.1 Update

The `update` operation (see code in figure A.8) takes a Form object as parameter and delivers a Form object as result. The Form object given as the parameter of the operation serves as the pattern: every key/value pair from the parameter Form object is copied to the actual Form object. Existing keys and values are overwritten in the actual Form object. The resulting Form object has the same keys/values as the parameter Form object plus the keys/values from the actual Form object that were not overwritten.

```
/** Updates this form with the contents of another one.
  * The existing contents are overwritten by those of the
  * form given as parameter.
  * The actual form changes its value! */
public Form update(Form form) {
  // iterate through the parameter form and replace (put) all
  // the items in the actual form (this).
  for (HashMapIterator i = form.begin(); !i.equals(form.end());
       i.advance()) {
    Object o = (Object)this.put(i.key(), i.value());
  }
  return this;
}
```

Figure A.8: The `update` operation on forms

Note, that as a side-effect of this implementation the Form object that the `update` operation is called on changes its value to the resulting Form object. Thus, `form1.merge(form2)` and `form2.update(form1)` are not the same, although the resulting Form object has the same keys and values.

A.6.2 Merge

The `merge` operation (see code in figure A.9) takes a Form object as parameter and delivers a Form object as result. It is almost specular to the `update` operation described before. Thus the `update` operation is used within the `merge` operation to prevent duplicated code. The only difference is that `merge` does not change the original Form objects. The resulting Form object has the same keys/values as the actual Form object plus the keys/values from the parameter Form objects that were not overwritten.

```
/** Merges this form with the contents of another one.
  * The existing contents are not overwritten. */
public Form merge(Form form) {
  Form result = new Form(form);              // This copy prevents the parameter form
  return result.update(this);                // from actually being changed.
}
```

Figure A.9: The `merge` operation on forms

Appendix B

Code Samples

In this chapter we present some code excerpts from the examples we implemented with APROCO. All the code presented in this thesis is freely available at the author's web page:
URL: `http://www.iam.unibe.ch/~dkuehni/aproco.html`.

B.1 Coordination Agents

B.1.1 Multicast Agent

The multicast agent is introduced in section 3.7 and used in three examples: the multicast example (section 3.7), the observer example (section 5.2), and the electronic vote example (section 5.3).

```
package aproco.examples.multicast;

import jada.*;
import aproco.lib.Form;
import java.util.*;

/** <b>Multicast Agent</b><p>
 *
 * This agent multicasts some forms into the receivers private
 * dataspaces. The forms it will multicast are parameters for
 * the multicast agent. It is simply an array of forms that
 * it should listen to. <p>
 *
 * The multicast agent needs the forms that it has to check
 * for as array in the parameters.
 */
public class MulticastAgent implements Runnable {

  ObjectSpace globalspace, agentspace;
  Form[] toListen;                              // The array of forms it has to multicast.
  Form listOfClients;
  Vector clientInfo;
  int numForms;                                 // The amount of forms in the array toListen.
```

```
public MulticastAgent(ObjectSpace globalDataSpace,
                      ObjectSpace agentDataSpace,
                      Form formsToListen[],
                      int numberOfForms) {
  globalspace = globalDataSpace;
  agentspace = agentDataSpace;
  toListen = formsToListen;
  numForms = numberOfForms;
  clientInfo = new Vector();
  listOfClients = new Form();
  listOfClients.put("Type", "ListOfClients");
}

/** Member class for the threads to listen for a particular
  * form in a data space.
  */
private class WaitingThread implements Runnable {

  Form form;                                    // The form to listen to.
  ObjectSpace ospace;                           // The data space that is has to listen in.

  private WaitingThread(ObjectSpace os, Form f) {
    ospace = os;
    form = f;
  }

  public void run() {
    while (true) {
      Tuple t = (Tuple)ospace.in(new Tuple(form));
      System.out.println("Multicast agent got something");
      Form g = (Form)t.getItem(0);
      multicastForm(g);
    }
  }
}

/** Multicasts the given form to the registered observers.
  * The registered observers are checked each time before the
  * actual multicasting is performed.
  */
public void multicastForm(Form f) {
  Tuple t = (Tuple)globalspace.read(new Tuple(listOfClients));
  Form g = (Form)t.getItem(0);
  clientInfo = (Vector)g.get("List");         // Update the list of receivers.
  for (Enumeration e = clientInfo.elements(); e.hasMoreElements();)
                                              // Performs the multicast.
    ((ObjectSpace)e.nextElement()).out(new Tuple(f));
}

/** Checks the agent data space for forms and multicasts them to
    the receivver's private data spaces.
 */
public void run() {
```

```
    System.out.println("Multicasting agent is working");
    System.out.print("Multicasting agent listens to forms: ");
    for (int i = 0; i < numForms; i++) {
      System.out.print(toListen[i].toString());
    }
    System.out.println();
    // check for forms to multicast, start a thread for each form
    for (int i = 0; i < numForms; i++) {
      new Thread(new WaitingThread(agentspace, toListen[i])).start();
    }
  }
}
```

B.1.2 Fault Tolerance Agent

The fault tolerance agent is introduced in section 5.1 and used in two examples: the fault tolerance service (section 5.1) and the administrator / worker example (section 5.4).

```
package aproco.examples.faulttolerance;

import jada.*;
import aproco.lib.Form;
import com.objectspace.jgl.*;                    // The JGL class library for HashMap.

/** <b>Fault Tolerance Agent</b><p>
 *
 * The fault tolerance agent checks requests and answers that are
 * transported from one data space into another. It uses a default
 * timeout that it associates with every request. This timeout is
 * checked. If it ran out, the request is repeated exactly once again
 * with a timeout. If it ran out again, an error message for the client
 * is created. <p>
 * If the server agent was only too slow and an answer for the original
 * and the repeated request is sent, only one is transported by the
 * fault tolerance agent.<p>
 *
 * The agent maintains a list of open requests and their timeouts.<p>
 *
 * Improvements: the agent could adjust the timeouts to limit unnecessarily
 * repeated requests.
 */
public class FaultTolerance implements Runnable {

  public static long delay       = 500;          // Delay for sleep in millis.
  public static long def_timeout = 5000;         // Default timeout in millis.

  ObjectSpace clientos, serveros;
  HashMap requests;                              // For open requests and their timeouts.
  int cid, sid;                                  // The client ID and the ID of this service.
  String type;
  Tuple clientt, servert;
```

```
  Form f, request, answer, clientf, serverf;

  public FaultTolerance(ObjectSpace clientSpace,
ObjectSpace serverSpace) {
    clientos = clientSpace;
    serveros = serverSpace;
    requests = new HashMap();
    request = new Form();
    request.put("Type", "Request");
    answer = new Form();
    answer.put("Type", "Answer");
  }

  /** This member class holds the abstraction for the timeout
   * value plus the information, if it was a repeated request.
   */
  class Timeout {
    long timeout;
    boolean repeat;                          // Indicates whether this request was repeated.

    Timeout(long to, boolean rp) {
      timeout = to;
      repeat = rp;
    }
  }

  /** Inserts the captured request along with the actual time into
   * the local tuple space. <p>
   * Only adds new requests, that are not already in the local
   * timeout list.
   */
  void insertRequest(Form f, boolean repeat) {
    long timeout = def_timeout + System.currentTimeMillis();
    Timeout t = (Timeout)requests.add(f, new Timeout(timeout, repeat));
                                                 // Returns null, if this is a new key!
  }

  /** Checks if there is a request stored in the HashMap
   * that corresponds to the just captured answer. <p>
   * Deletes the open request when there is one otherwise prints out
   * an error message. <p>
   *
   * @return If the corresponding request was successfully removed
   *         it returns the form that was given as parameter. If the
   *         request was not found, it returns null.
   */
  Form answerCheck(Form f) {
    Form r = new Form(request);
    r.put("ClientID", (Integer)f.get("ClientID"));
    r.put("ServiceID",(Integer)f.get("ServiceID"));
    r.put("Parameters",(String)f.get("Parameters"));
    Timeout t = (Timeout)requests.remove(r);
    if (t == null) {
      System.out.println("No corresponding request found!");
```

```
      return null;
    }
    else return f;
  }

  /** Checks if there is an entry in the requests that has an
   * expired timeout. If such an entry exists, it will be
   * removed. */
  void timeoutCheck() {
    long time = System.currentTimeMillis();
    long timeout;
    Form f;
    for (HashMapIterator i = requests.begin(); !i.equals(requests.end());
                  i.advance()) {
      timeout = ((Timeout)i.value()).timeout;
      if (time > timeout) {
f = (Form)i.key();
Timeout t = (Timeout)requests.remove(f);
if (t.repeat) {                                  // Repeated request, create error answer.
  System.out.println("Timeout exceeded, request removed!");
  createErrorAnswer(f);
  System.out.println("Timeout exceeded, error answer created");
}
else {                                           // Repeat answer to enable another server to answer.
  clientos.out(new Tuple(f));
  insertRequest(f, true);
  System.out.println("Timeout exceeded, request repeated");
}
      }
    }
  }

  /** Creates an error answer for the client. */
  void createErrorAnswer(Form f) {
    Form r = new Form(answer);
    r.put("ClientID", (Integer)f.get("ClientID"));
    r.put("ServiceID",(Integer)f.get("ServiceID"));
    r.put("Result", "Error happened");
    clientos.out(new Tuple(r));                  // Put Error answer into client space.
  }

  /** Continuously checks the client and the server data spaces for
    * new reqests respectively answers and checks for timeouts. */
  public void run() {
    System.out.println("Fault Tolerance is working");
    while (true) {
      do {
        try { Thread.sleep(delay); }
        catch (InterruptedException e) { };
        timeoutCheck();
        clientt = (Tuple)clientos.in_nb(new Tuple(request));
        servert = (Tuple)serveros.in_nb(new Tuple(answer));
      } while (clientt == null && servert == null);
      if (clientt != null) {
```

```
        clientf = (Form)clientt.getItem(0);
        insertRequest(clientf, false);
        serveros.out(new Tuple(clientf));       // Put into server space.
      }
      if (servert != null) {
        serverf = (Form)servert.getItem(0);
        f = answerCheck(serverf);
        if (f != null) clientos.out(new Tuple(serverf));
        // Only gets delivered if a corresponding request could be found!
      }
    }
  }
}
```

B.1.3 Registration Agent

The registration agent is introduced in section 5.2 and used in two examples: the observer example (section 5.2) and the electronic vote example (section 5.3).

```
package aproco.examples.observer

import java.util.*;
import jada.*;
import aproco.lib.Form;

/**
 * <b>Registration Agent</b>
 *
 * The registration agent listens to new client agents registering with
 * the medium over the global configuration data space. It maintains a
 * list of the currently registered client agents as form in the global
 * configuration data space.
 */
public class RegistrationAgent implements Runnable {

  ObjectSpace globalspace;
  Form registerClient, deregisterClient, registerCoord, registerNotification,
    listOfClients;
  Vector clientInfo, coordAgents;

  public RegistrationAgent(ObjectSpace global) {
    globalspace = global;
    registerClient = new Form();
    registerClient.put("Type", "RegisterClient");
    deregisterClient = new Form();
    deregisterClient.put("Type", "DeregisterClient");
    registerCoord = new Form();
    registerCoord.put("Type", "RegisterCoordAgent");
    registerNotification = new Form();
    registerNotification.put("Type", "RegisterNotification");
    listOfClients = new Form();
```

```
    listOfClients.put("Type", "ListOfClients");
    coordAgents = new Vector();
  }


  /** Updates the list of clients stored in the global information
   * space.
   */
  void updateClientList(Form clientf) {
    Tuple t = (Tuple)globalspace.in_nb(new Tuple(listOfClients));
    if (t != null) {
      Form g = (Form)t.getItem(0);
      clientInfo = (Vector)g.get("List");
    }
    ObjectSpace handle = (ObjectSpace)clientf.get("Handle");
    if (!clientInfo.contains(handle)) {         // Duplicate entries are not allowed.
      clientInfo.addElement(handle);
    }
    Form f = new Form(listOfClients);
    f.put("List", clientInfo);
    globalspace.out(new Tuple(f));
  }

  /** Removes the clients private data space reference from the list
    * of registered clients.
    */
  void removeThisClient(Form clientf) {
    Tuple t = (Tuple)globalspace.in_nb(new Tuple(listOfClients));
    if (t != null) {
      Form g = (Form)t.getItem(0);
      clientInfo = (Vector)g.get("List");
      ObjectSpace handle = (ObjectSpace)clientf.get("Handle");
      boolean dontcare = clientInfo.removeElement(handle);
      g.put("List", clientInfo);                // Overwrites the old value.
      globalspace.out(new Tuple(g));
    }
  }

  /** Updates the list of interested coordination agents. This list is
   * stored internally.
   */
  void updateCoordList(Form coordf) {
    String name = (String)coordf.get("Name");
    if (!coordAgents.contains(name)) {          // Duplicate entries are not allowed.
      coordAgents.addElement(name);
    }
  }

  /** Puts notations for the registered coordination agents into the
    * global configuration data space on every newly created client
    * agent. This information includes the data space references of
    * the client agent's private data spaces.
    */
  void notifyCoordAgents() {
```

```
    Tuple t = (Tuple)globalspace.read(new Tuple(listOfClients));
    Form g = (Form)t.getItem(0);
    Form notification = new Form(registerNotification);
    notification.put("List", (Vector)g.get("List"));
    for (Enumeration e = coordAgents.elements(); e.hasMoreElements();) {
      Form n = new Form(notification);
      n.put("Name", (String)e.nextElement());
      globalspace.out(new Tuple(n));
    }
  }

  /** Waits for new client agents or coordination agent to register or
    * deregister with APROCO.
    */
  public void run() {
    System.out.println("Registration Agent is working");
    Form finit = new Form(listOfClients);
    finit.put("List", new Vector());
    globalspace.out(new Tuple(finit));
    new Thread(new Runnable() {
      // Thread waiting for new clients to register.
      public void run() {
        while (true) {
          Tuple t = (Tuple)globalspace.in(new Tuple(registerClient));
          System.out.println("Client has registered");
          Form g = (Form)t.getItem(0);
          updateClientList(g);
          notifyCoordAgents();
        }
      }
    }).start();
    new Thread(new Runnable() {
      // Thread waiting for coordination agents to register.
      public void run() {
        while (true) {
          Tuple t = (Tuple)globalspace.in(new Tuple(registerCoord));
          Form g = (Form)t.getItem(0);
          updateCoordList(g);
        }
      }
    }).start();
    new Thread(new Runnable() {
      // Thread waiting for clients to deregister.
      public void run() {
        while (true) {
          Tuple t = (Tuple)globalspace.in(new Tuple(deregisterClient));
          Form g = (Form)t.getItem(0);
          removeThisClient(g);
        }
      }
    }).start();
  }
}
```

B.1.4 Authentication Agent

The authentication agent is introduced in section 5.3 and used in the electronic vote example (section 5.3).

```
package aproco.examples.vote;

import jada.*;
import aproco.lib.Form;
import java.util.*;

/** <b>Authentication Agent</b><p>
 *
 * This agent checks the registered voter's private data spaces for forms
 * and passes them to the agent data space. The authentication agent adds
 * and ID to each form originating from the same private data space to
 * allow the other coordination agents to identify its origin.
 */
public class AuthenticationAgent implements Runnable {

  ObjectSpace subjectspace, globalspace;
  Form registerCoord, registerNotification, listOfClients, request, answer;
  Hashtable voterSpaces;                        // Private data spaces and IDs.
  Vector clientInfo;                            // The voter's private data spaces.
  int idCount;                                  // Base of the ID

  public AuthenticationAgent(ObjectSpace globalDataSpace,
                             ObjectSpace agentDataSpace) {
    subjectspace = agentDataSpace;
    globalspace = globalDataSpace;
    voterSpaces = new Hashtable();
    clientInfo = new Vector();
    idCount = 1;
    registerCoord = new Form();
    registerCoord.put("Type", "RegisterCoordAgent");
    registerCoord.put("Name", "Authentication");
    registerNotification = new Form();
    registerNotification.put("Type", "RegisterNotification");
    registerNotification.put("Name", "Collector");
    listOfClients = new Form();
    listOfClients.put("Type", "ListOfClients");
    request = new Form();
    request.put("Type", "Request");
  }

  /** Checks if the registered voters are in the internal list and assigns
    * and ID to them.
    */
  synchronized void updateListOfVoters() {
    for (Enumeration e = clientInfo.elements(); e.hasMoreElements();) {
      ObjectSpace handle = (ObjectSpace)e.nextElement();
      if (!voterSpaces.containsKey(handle))
        voterSpaces.put(handle, new Integer(idCount++));
```

```
    }
    for (Enumeration e = voterSpaces.keys(); e.hasMoreElements();) {
      ObjectSpace handle = (ObjectSpace)e.nextElement();
      if (!clientInfo.contains(handle))
        voterSpaces.remove(handle);
    }
  }

  /** Continously checks all the private voter spaces for their votes
    * and newly initiated votes.
   */
  public void run() {
    System.out.println("Authentication agent is working");
    // Register with the medium to get notification about new clients.
    globalspace.out(new Tuple(registerCoord));
    new Thread(new Runnable() {
      // Thread waiting for register notification, updates this information.
      public void run() {
        while (true) {
          Tuple t = (Tuple)globalspace.in(new Tuple(registerNotification));
          Form g = (Form)t.getItem(0);
          clientInfo = (Vector)g.get("List");
          updateListOfVoters();
        }
      }
    }).start();
    new Thread(new Runnable() {
      // Thread checking the private data space of the observers for requests.
      public void run() {
        Tuple t;
        Integer id;
        do {                                          // Initialize the list; wait for a list to appear.
          t = (Tuple)globalspace.read_nb(new Tuple(listOfClients));
          if (t != null) {
            Form g = (Form)t.getItem(0);
            clientInfo = (Vector)g.get("List");
            updateListOfVoters();
          }
          try { Thread.sleep((long)(Math.random()*200)); }
          catch (InterruptedException e) { };
        } while(t == null);
        // The main loop
        while (true) {
          for (Enumeration e = voterSpaces.keys(); e.hasMoreElements();) {
            ObjectSpace handle = (ObjectSpace)e.nextElement();
            t = (Tuple)handle.in_nb(new Tuple(request));
            if (t != null) {
              Form f = (Form)t.getItem(0);
              // Add this voter's ID to the request.
              id = (Integer)voterSpaces.get(handle);
              f.put("VoterID", id);
              // Put the request into the subject's space.
              subjectspace.out(new Tuple(f));
            }
```

```
          }
          try { Thread.sleep(200); }
          catch (InterruptedException e) { };
        }
      }
    }).start();
  }
}
```

B.2 A Complete Example

We present the Java code of the fault tolerance service example presented in section 5.1. Although this is admittedly not the most exciting example, it shows the main properties of APROCO as well as its simplicity to set up client agents and coordination agents. We present all parts except the fault tolerance agent, because its code can be found in the previous section.

B.2.1 Fault Tolerance Service

This is the main routine of the fault tolerance service example. It first initializes the required data spaces and starts all the agents that constitutes the whole application. The data spaces are parameters for these agents. The agents typically have more parameters, in this example for their identification numbers.

```
package aproco.examples.faulttolerance;

import jada.*;

/**
 * <b><i>Fault Tolerance Service</b></i><p>
 *
 * The example consists of a client that wants a certain serivce done
 * from a unknown server. This server or a multitude of them may fail
 * to deliver the desired service. <p>
 * The client has to be aware of the possibility of failure (some form
 * of error handling).<p>
 *
 * Start: java aproco.examples.faulttolerance.FaultToleranceService <p>
 *
 * 21.9.98, Daniel Kuehni <p>
 *
 * @see Server
 * @see TestClient
 * @see FaultTolerance
 */
public class FaultToleranceService {

  public static void main(String args[]) {
    // Create new (local) object spaces.
```

```
    ObjectSpace clientspace = new ObjectSpace();
    ObjectSpace serverspace = new ObjectSpace();
    // Create a server.
    new Thread(new Server(17, serverspace)).start();
    // Create another server with the same ID.
    new Thread(new Server(17, serverspace)).start();
    // Create some clients.
    new Thread(new TestClient(1, clientspace)).start();
    new Thread(new TestClient(2, clientspace)).start();
    new Thread(new TestClient(3, clientspace)).start();
    // Create the fault tolerance agent.
    new Thread(new FaultTolerance(clientspace, serverspace)).start();
  }
}
```

B.2.2 Server Agent

The server agent listens for service requests addressed to its service identification. It is unstable and can fail to deliver the desired answer. To simulate failure the server agent starts an internal thread that is terminating the main thread after a randomly chosen time.

```
package aproco.examples.faulttolerance;

import jada.*;
import aproco.lib.Form;

/** Server thread for fault tolerance service example. <p>
 * The server is unstable and can fail at any time without automatic
 * recovery and eventual loss of the client's request.
 */
public class Server implements Runnable {

  ObjectSpace os;
  int cid, sid;                                  // The service ID that this server operates.
  boolean running;
  String params, result;


  public Server(int serviceID, ObjectSpace serverSpace) {
    sid = serviceID;
    os = serverSpace;
    running = true;
  }

  public String performService(String params) {
    // Simulates the calculattion of a result with the supplied parameters.
    return "amazing result";                     // Not really...
  }

  /** Internal routine to shut down the server randomly. */
  private void initKillServer() {
```

```
    (new Thread() {
      public void run() {
        try { Thread.sleep((long)(Math.random()*10000)); } // 10 secs.
        catch (InterruptedException e) { };
        System.out.println("Server with ID "+sid+" is going down");
        running = false;                        // Tell the server to stop.
      }
    }).start();
  }

  /** The server continuously looks for requests for his service ID
   * and delivers answers back with the same client ID as in the
   * request.
   */
  public void run() {
    Tuple t;
    System.out.println("Server with ID "+sid+" is waiting for requests");
    initKillServer();
    while (running) {
      // Block waiting for a request that matches service ID.
      Form form = new Form();
      form.put("Type", "Request");
      form.put("ServiceID", new Integer(sid));
      t = (Tuple)os.in(new Tuple(form));
      Form f = (Form)t.getItem(0);
      if (running) {
        params = (String)f.get("Parameters");
        result = performService(params);        // Perform the service.
        // Create the result tuple and put it into the tuple space.
        Form res = new Form(f);
        res.put("Type", "Answer");
        res.put("Result", new String(result));
        os.out(new Tuple(res));
      }
    }
  }
}
```

B.2.3 Client Agent

The client agent puts requests for a particular service into a data space and waits for corresponding answers addressed to it. It has to be able to handle error messages instead of valid results.

```
package aproco.examples.faulttolerance;

import jada.*;
import aproco.lib.Form;

/** Client thread for fault tolerance service example.<p>
 * This version is sending a new request every 10 seconds and block
 * waits for the result. Parameters and results are strings.
```

```
 */
public class TestClient implements Runnable {

  ObjectSpace os;
  int cid;                                          // The client's ID.
  Form share, request, answer;


  public TestClient(ObjectSpace ospace) {
    this(0, ospace);
  }

  public TestClient(int id, ObjectSpace ospace) {
    cid = id;
    os = ospace;
    share = new Form();
    share.put("ClientID", new Integer(cid));
    request = new Form(share);
    request.put("Type", "Request");
    answer = new Form(share);
    answer.put("Type", "Answer");
  }

  /** The client puts a service request into the tuple space. */
  public void requestService(int sid, String params) {
    Form form = new Form(request);
    form.put("ServiceID", new Integer(sid));
    form.put("Parameters", new String(params));
    os.out(new Tuple(form));
  }

  /** Block waits for an answer that matches the same service ID and
   * the client's ID. */
  public String getAnswer(int sid) {
    Form form = new Form(answer);
    form.put("ServiceID", new Integer(sid));
    Tuple t = (Tuple)os.in(new Tuple(form));
    Form f = (Form)t.getItem(0);
    return (String)f.get("Result");
  }

  public void run() {
    while(true) {
      try { Thread.sleep((long)(Math.random()*1000)); }
      catch (InterruptedException e) { };
      System.out.println("Client " + cid + " makes service request");
      requestService(17, "no relevant params");
      String result = getAnswer(17);
      System.out.println("Client " + cid + " got answer: " + result);
    }
  }
}
```

Bibliography

[AB92] Mehmet Aksit and Lodewijk Bergmans. Obstacles in Object-Oriented Software Development. In *OOPSLA '92*, pages 341–358, Vancover, Canada, 1992.

[AF+94] Gul Agha, Svend Frølund, et al. *Research Directions in Concurrent Object-Oriented Programming*, chapter Abstraction and Modularity Mechanisms for Concurrent Computing, pages 3–21. MIT Press, 1994.

[BCG97] Robert Bjornson, Nicholas Carriero, and David Gelernter. From Weaving Threads to Untangling the Web: A View of Coordination from Linda's Perspective. In D. Garlan and D. Le Metayer, editors, *Coordination Languages and Models (COORDINATION '97)*, LNCS 1282, pages 1–17. Springer-Verlag, 1997.

[BLM93] J. P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[Blo79] Toby Bloom. Evaluating Synchronization Mechanisms. In *Seventh ACM Symposium on Operating Systems Principles*, 1979.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stad. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley, 1996.

[CA94] Christian J. Callsen and Gul Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, 21:289–300, 1994.

[CDK94] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, second edition, 1994.

[CG90] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course*. The MIT Press, 1990. Second Printing 1991.

[Cia96] Paolo Ciancarini. Coordination Models and Languages as Software Integrators. *ACM Computing Surveys*, 28(2):300–302, 1996.

[Cia97] Paolo Ciancarini. Coordination Models, Languages, Architectures, and Applications. http://www.cs.unibo.it/~cianca/wwwpages/coord_ToC.html, October 1997. Slides from lectures at University of Leuven, Belgium, Feb. 97.

[CR96] Paolo Ciancarini and Davide Rossi. Jada: Coordination and Communication for Java Agents. In *Sec. Intl. Workshop on Mobile Object Systems: Towards the Programmable Internet (MOS'96)*, LNCS 1222, pages 213–228. Springer-Verlag, July 1996.

[DNO97] Enrico Denti, Antonio Natali, and Andrea Omicini. Programmable Coordinarion Media. In D. Garlan and D. Le Metayer, editors, *Coordination Languages and Models (COORDINATION '97)*, LNCS 1282, pages 274–288. Springer-Verlag, 1997.

[FA93] Svend Frølund and Gul Agha. A Language Framework for Multi-Object Coordination. In *European Conf. on O.O. Programming (ECOOP'93)*, LNCS 707, pages 346–360. Springer-Verlag, 1993.

[FG96] Stan Franklin and Art Graesser. Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents. In *Third Intl. Workshop on Agent Theories, Architectures, and Languages*, pages 21–35. Springer-Verlag, 1996.

[Fla97] David Flanagan. *Java in a Nutshell, Second Edition*. O'Reilly, 1997.

[GC92] David Gelernter and Nicholas Carriero. Coordination Languages and their Significance. *Comm. of the ACM*, 35(2):96–107, February 1992.

[Gel85] David Gelernter. Generative Communication in Linda. *ACM Trans. Programming Languages and Systems*, 7(1):80–112, 1985.

[Gen81] Morven Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. *Software - Practice and Experience*, 11:435–466, 1981.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, School of Computer Science, Carnegie Mellon University, 1994. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/able/ftp/intro-softarch.ps.

[Gün98] Manuel Günter. Explicit Connectors for Coordination of Active Objects. Master's thesis, Inst. of Computer Science and Applied Mathematics, University of Berne, March 1998.

[Ham97] Graham Hamilton. *Java Beans 1.01*. Sun Microsystems, Mountain View, USA, version 1.01 edition, July 1997. http://splash.javasoft.com/beans/spec.html.

[Hol97] Tom Holvoet. *An Approach for Open Concurrent Software Development*. PhD thesis, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, December 1997.

[IBM98a] IBM Almaden Research Center, San Jose, CA, USA. *IBM T Spaces User's Guide*, 1998. http://www.almaden.ibm.com/cs/TSpaces/.

[IBM98b] IBM Research, Tokyo, Japan. *Aglets Software Development Kit*, 1998. http://www.trl.ibm.co.jp/aglets/.

[Kie97] Thilo Kielmann. *Objective Linda: A Coordination Model for Object-Oriented Parallel Programming.* PhD thesis, Dept. of Electrical Engineering and Computer Science, University of Siegen, Germany, 1997. Shaker Verlag, Aachen.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, LNCS 1241, pages 220–242., Jyvaskyla, Finland, June 1997. Springer-Verlag.

[LAN98] Markus Lumpe, Franz Acherman, and Oscar Nierstrasz. An extensible language for Compostion. Paper, Software Composition Group, University of Berne, Switzerland, September 1998. Submitted for publication.

[Lea97] Doug Lea. *Concureent Programming in Java: Design Principles and Patterns*. The Java Series. Addison-Wesley, 1997.

[Mal88] Thomas W. Malone. What is Coordination Theory. Working paper no. 2051-88, MIT Sloan School of Management, Cambridge, Mass., 1988.

[MC94] Thomas W. Malone and Kevin Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87–119, March 1994.

[Mor81] W. Morris. *The American Heritage Dictionary of the English Language*. Houghton Mifflin Company, 1981.

[MU97] Naftaly H. Minsky and Victoria Ungureanu. Regulated Coordination in Open Distributed Systems. In D. Garlan and D. Le Metayer, editors, *Coordination Languages and Models (COORDINATION '97)*, LNCS 1282, pages 81–97. Springer-Verlag, 1997.

[ND95] Oscar Nierstrasz and Laurent Dami. Component-Oriented Software Technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.

[Obj97] ObjectSpace Inc., Dallas, TX, USA. *The Generic Collection Library for Java(tm), Version 3.1*, 1997. http://www.objectspace.com/jgl/.

[Obj98] ObjectSpace Inc., Dallas, TX, USA. *ObjectSpace Voyager(tm)*, 1998. http://www.objectspace.com/voyager/.

[PA98] George A. Papadopoulos and Farhad Arbab. *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, chapter Coordination Models and Languages. Academic Press, August 1998.

[Pap95] Michael Papathomas. Concurrency in Object-Oriented Programming Languages. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 31–68. Prentice Hall, 1995.

[Rog97] Dale Rogerson. *Inside COM*. Microsoft Press, 1997.

[RW96] Antony Rowstron and Alan Wood. Solving the Linda Multiple rd Problem. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION '96)*, LNCS 1061, pages 357–367. Springer-Verlag, 1996.

[Sci94] Scientific Computing Accociates, New Haven, CT. *Paradise: User's Guide and Reference Manual*. 1994.

[Sin92] B. Singh. Interconnected Roles (IR): A Coordinated Model. Technical Report CT-84-92, Microelectronics and Computer Technology Corp., Austin, TX, 1992.

[SO+96] Marc Snir, Steve Otto, et al. *MPI: The Complete Reference*. MIT Press, 1996.

[Ste90] Guy L. Steele. *Common Lisp The Language, Second Edition*. Digital Press, 1990.

[Sun98] Sun Microsystems Inc., Mountain View, CA, USA. *JavaSpaces(tm) Specification*, 1.0 beta edition, July 1998. http://java.sun.com/products/javaspaces/.

[Tic97] Sander Tichelaar. A Coordination Component Framework for Open Distributed Systems. Master's thesis, Dept. of Computer Science, University of Groningen, May 1997.

[Tsi89] Dennis Tsichritzis. Object-Oriented Development for Open Systems. In *Information Processing 89 (Proceedings IFIP'89)*, pages 1033–1040, 1989.

[vR97] Guido van Rossum. *Python Reference Manual*. Corporation for National Research Initiatives (CNRI), rel. 1.5 edition, December 1997.