Coordination of Distributed Systems

Distributed Systems / Paradigms
Sistemi Distribuiti / Paradigmi

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2017/2018

Next in Line. . .

Scenarios for Distributed Systems

Issues

concurrency / parallelism
- *multiple* independent activities / loci of control
- active *simultaneously*
- processes, threads, actors, active objects, agents. . .

distribution
- activities running on different and heterogeneous execution *contexts* (machines, devices, . . .)

social interaction
- *dependencies* among activities
- collective *goals* involving activities coordination / cooperation

environmental interaction
- interaction with external *resources*
- interaction within the *time-space* fabric
- interaction in *context*

Complexity & Interaction I

An essential source of complexity for computational systems is

interaction

[Goldin et al., 2006a]

The power of interaction [Wegner, 1997]

Interaction is a more powerful paradigm than rule-based algorithms for computer-based solving, overtiring the prevailing view that all computing is expressible as algorithms.

Complexity & Interaction II

Intelligence & interaction [Brooks, 1991]

Real computational systems are not rational agents that take inputs, compute logically, and produce outputs... It is hard to draw the line at what is intelligence and what is environmental interaction. In a sense, it does not really matter which is which, as all intelligent systems must be situated in some world or other if they are to be useful entities.

A conceptual framework for interaction [Milner, 1993]

... a theory of concurrency and interaction requires a new conceptual framework, not just a refinement of what we find natural for sequential [algorithmic] computing.

Complexity & Interaction III

Interactive computing [Wegner and Goldin, 1999]

- *finite computing agents* that interact with an environment are shown to be more expressive than Turing machines according to a notion of expressiveness that measures problem-solving ability and is specified by observation equivalence
- *sequential interactive models* of objects, agents, and embedded systems are shown to be more expressive than algorithms
- *multi-agent* (distributed) *models of coordination*, collaboration, and true concurrency are shown to be more expressive than sequential models

Complexity & Interaction IV

Basically, where does complexity come from?

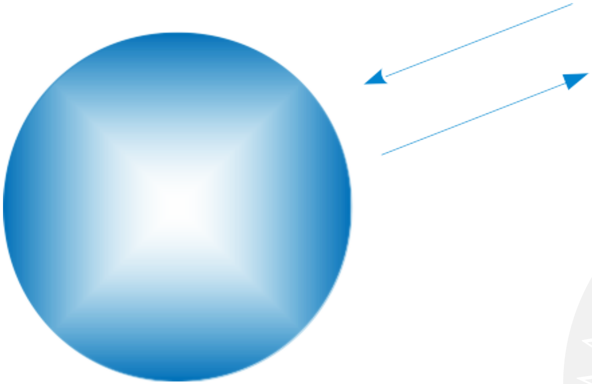
- events in a sequential component are *totally ordered*
- as soon as we combine components in a concurrent system (*distribution in time*), they are *no* longer totally *ordered*
- as soon as we combine components in a distributed system (*distribution in space*), interaction occurs in different contexts
- interaction make the overall system essentially unpredictable

? why is this *not a problem*?

! the range of behaviours that an interactive system can exhibit is typically larger than non-interactive systems

→ more behaviours means more expressiveness

Components of an Interacting System I



Computational process with *input* and *output*

Components of an Interacting System II

What is a component of an interacting system?

- a computational abstraction characterised by
  - an independent computational activity
  - I/O capabilities
- two independent dimensions
  - elaboration / *computation*
  - *interaction*
- nothing else

openness	no hypothesis on the component's life & behaviour
distribution	no hypothesis on the component's location & motion
heterogeneity	no hypothesis on the component's nature & structure

Basics of Interaction

Component model

A simple component exhibits

computation	inner behaviour of a component
interaction	observable behaviour of a component as *input* and *output*

Coupling across component's boundaries

- control?
- information
- time & space—internal / computational vs. external / physical

Information-driven interaction

output	shows part of its state outside
input	bounds a portion of its own state to the outside

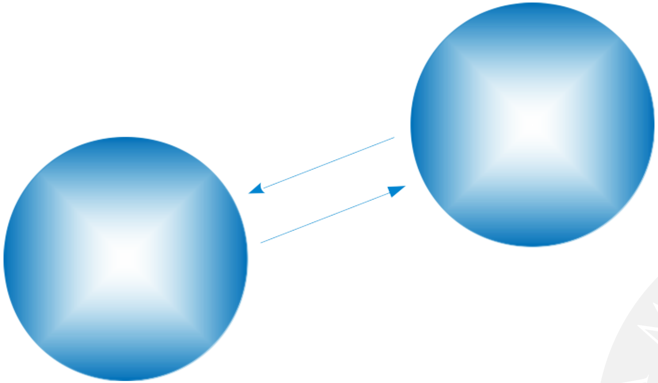
(Interacting) Computational System [Goldin et al., 2006b] I

Computational system

In a computational system, two or more computational processes

- *behave* (by computing), and
- *work together* (by interacting)

(Interacting) Computational System [Goldin et al., 2006b] II

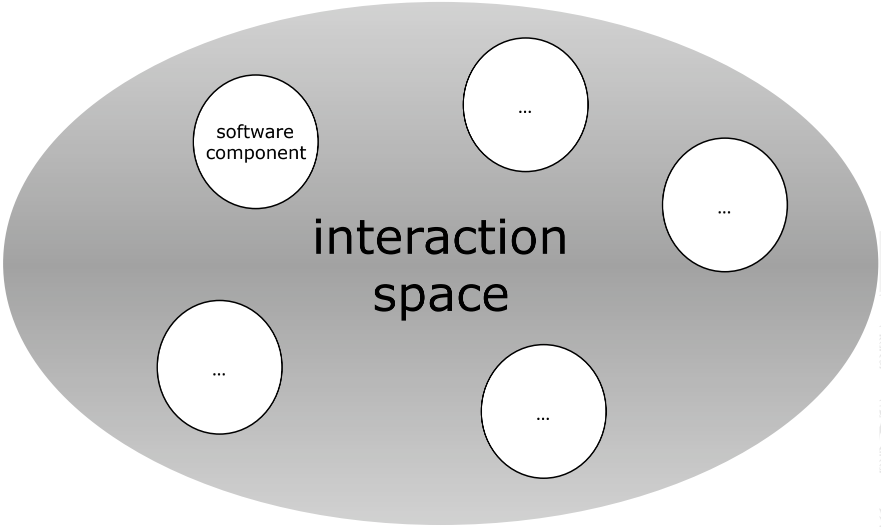


Basic interacting computational system

Next in Line. . .

Focus on. . .

Interacting System



Coordination in Distributed Programming I

Coordination model as a glue

A coordination model is the glue that binds separate activities into an ensemble

[Gelernter and Carriero, 1992]

Coordination model as an agent interaction framework

A coordination model provides a framework in which the interaction of active and independent entities called agents can be expressed

[Ciancarini, 1996]

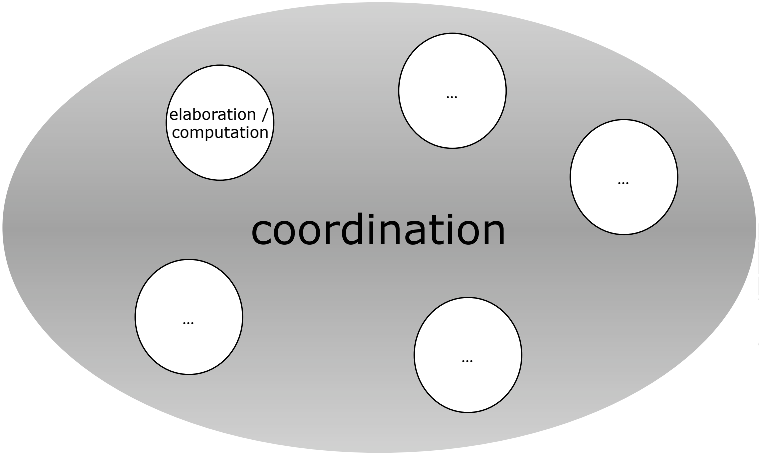
Coordination in Distributed Programming II

Issues for a coordination model

A coordination model should cover the issues of creation and destruction of agents, communication among agents, and spatial distribution of agents, as well as synchronization and distribution of their actions over time

[Ciancarini, 1996]

What is Coordination?



A New Perspective over Computational Systems

Programming languages

- interaction as an *orthogonal* dimension
- *languages* for interaction / *coordination*

Software engineering

- interaction as an *independent design* dimension
- coordination *patterns*

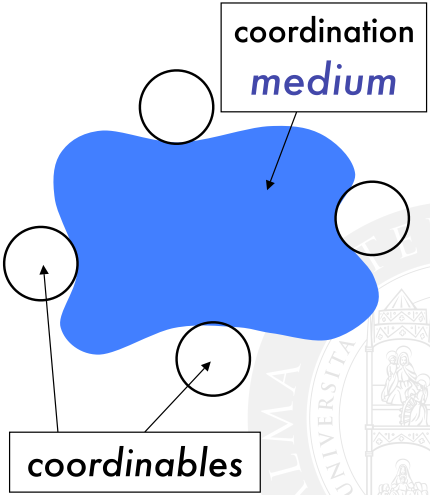
Artificial intelligence

- interaction as a new *source for intelligence*
- *social intelligence*

Coordination: Sketching a Meta-model

The *medium of coordination*

- "fills" the interaction space
- enables / promotes / governs the admissible / desirable / required interactions among the interacting entities
- according to some *coordination laws*
  - enacted by the behaviour of the medium
  - defining the semantics of coordination



Coordination: A Meta-model [Ciancarini, 1996]

A constructive approach

Which are the components of a coordination system?

coordination entities entities whose mutual interaction is ruled by the model, also called the *coordinables*

coordination media abstractions enabling and ruling interaction among coordinables

coordination laws laws ruling the observable behaviour of coordination media and coordinables, and their interaction as well

Coordinables

Original definition [Ciancarini, 1996]

These are the entity types that are coordinated. These could be Unix-like processes, threads, concurrent objects and the like, and even users.

examples	processes, threads, objects, human users, agents, . . .
focus	observable behaviour of the coordinables
question	are we anyhow concerned here with the internal machinery / functioning of the coordinable, in principle?

$\rightarrow$ *this issue will be clear when comparing* LINDA *&* TuCSoN *agents*

Coordination Media

Original definition [Ciancarini, 1996]

These are the media making communication among the agents possible. Moreover, a coordination medium can serve to aggregate agents that should be manipulated as a whole. Examples are classic media such as semaphores, monitors, or channels, or more complex media such as tuple spaces, blackboards, pipelines, and the like.

examples	semaphors, monitors, channels, tuple spaces, blackboards, pipes, . . .
focus	the core around which the components of the system are organised
question	which are the possible computational models for coordination media?

→ *this issue will be clear when comparing* LINDA *tuple spaces &* ReSpecT *tuple centres*

Coordination Laws I

Original definition [Ciancarini, 1996]

A coordination model should dictate a number of laws to describe how agents coordinate themselves through the given coordination media and using a number of coordination primitives. Examples are laws that enact either synchronous or asynchronous behaviors or exploit explicit or implicit naming schemes for coordination entities.

Coordination Laws II

- coordination laws rule the observable behaviour of coordination media and coordinables, as well as their interaction
    - a notion of (*admissible interaction*) *event* is required to define coordination laws
- the interaction events are (also) expressed in terms of
    - the communication language, as the syntax used to express and exchange data structures

  examples tuples, XML elements, FOL terms, (Java) objects, ...

    - the coordination language, as the set of the admissible interaction primitives, along with their semantics

  examples in/out/rd (LINDA), send/receive (channels), push/pull (pipes), ...

Focus on. . .

Basic Engineering Principles

Principles

abstraction
- problems should be faced / represented at the most suitable *level of abstraction*
- resulting abstractions should be *expressive* enough to capture the most relevant problems
- *conceptual integrity*

locality & encapsulation
- design abstractions should embody the solutions corresponding to the domain entities they represent

run-time vs. design-time abstractions
- incremental change / evolution
- on-line engineering [Fredriksson and Gustavsson, 2004]

Toward a Notion of Coordination Model

What do we ask to a coordination model?

- to provide high-level abstractions and powerful mechanisms for distributed system engineering
- coordination abstractions should deal with interaction
- to enable and promote the construction of *open, distributed, heterogeneous* systems
- to intrinsically add properties to systems *independently of components*
  - e.g. robustness, adaptiveness, intelligence, . . .
- by suitably shaping the interaction space

Examples of Coordination Mechanisms I

Message passing

- communication among peers
- no abstractions apart from message
- no limitations
  - the notion of *protocol* could be added as a coordination abstraction
- no intrinsic *model* of coordination
- any pattern of coordination can be superimposed [Deugo et al., 2001]
  - e.g., protocols as *coordination patterns* without dedicated software abstraction

Examples of Coordination Mechanisms II

Agent communication languages (ACL)

- goal: promote information exchange
- examples: Arcol, KQML
- standard: FIPA ACL [FIPA ACL, 2002]
- semantics: ontologies
- *enabling communication*
    - ACL *create* the space of inter-agent communication
    - they do *not* allow it to be *constrained*

Examples of Coordination Mechanisms III

Communication protocols

- goal: setting communication patterns outside agents
- examples / standard: FIPA / JADE protocols [FIPA ACL, 2002]
- *governing communication*
  - protocols *rule* the space of inter-agent communication
  - they *do* allow it to be *constrained*

Service-Oriented Architectures (SOA) [Erl, 2005]

- basic abstraction: service
- basic pattern: service request / response
- several standards
- very simple pattern of coordination

Examples of Coordination Mechanisms IV

Web Server

- basic abstraction: resource (REST/ROA)
- basic pattern: resource request / representation / response
- several standards
- again, a very simple pattern of coordination
- generally speaking, objects, HTTP, applets, JavaScript with AJAX, user interface
  - a *multi-coordinated* systems
  - "spaghetti-coordination", no value added from composition
- how can we fill the space of interaction to add value to systems?
  - so, how do we get *value from coordination*?

Examples of Coordination Mechanisms V

Middleware

goal to provide global properties across distributed systems

idea fill the space of interaction with abstractions and shared features

- interoperability, security, transactionality, ...

- middleware can contain abstractions of any sort, including coordination ones
  - e.g., JADE white & yellow pages services, which *inform* the space of agent interaction

  ? are they coordination abstractions?

  ?!? do we really care?

Enabling vs. Governing Interaction I

Enabling interaction

- e.g., ACL, SOA, . . .
- enabling communication
- enabling components interoperation
- no models for coordination of components
- *no rules* on what components should (not) say and do at any given moment, depending on what other components say and do, and on what happens inside and outside the system

Enabling vs. Governing Interaction II

Governing interaction

- e.g.

  JADE AMS no interaction before agents are registered

  FIPA protocols inter-agent communication should follow some pre-defined patterns

- in general, a model that does
  - rule what components should (not) say and do at any given moment
  - depending on what other components say and do, and on what happens inside and outside the system

Basic Question

Do we have a general model for coordination?

- do we have a general model for governing communication in distributed systems?
- do we have a general model for governing interaction in distributed systems?
- if we do
    - what technologies?
    - what methodologies?
    - what systems out of those?

Focus on. . .

Two Classes for Coordination Models

Control-oriented vs. data-oriented models

- control-driven vs. data-driven models
  [Papadopoulos and Arbab, 1998]

control-oriented focus on the *acts* of communication

data-oriented focus on the *information* exchanged during communication

- several surveys, some books [Omicini et al., 2001], no time enough here
- are these really *classes*?
  - ! actually, better to take this as a criterion to observe coordination models, rather than to separate them

Control-oriented Models I

Processes as black boxes

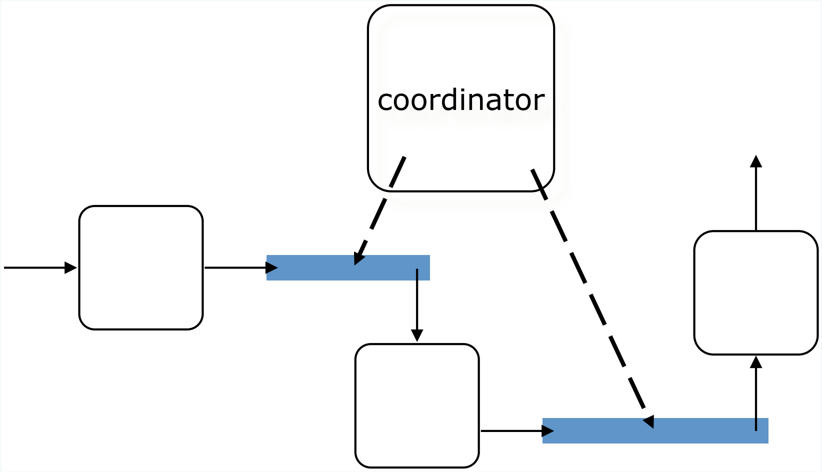
- I/O ports
- events & signals on state

Coordinators. . .

- . . . create coordinated processes as well as communication channels
- . . . determine and change the topology of communication
- hierarchies of coordinables / coordinators are possible

Control-oriented Models II

Coordinators as meta-level communication components



Control-oriented Models III

General features

- high flexibility, high control
- separation between communication / coordination and computation / elaboration
- examples
  - RAPIDE [Luckham et al., 1995]
  - Manifold [Arbab et al., 1993]
  - ConCoord [Holzbacher, 1996]
  - Reo [Arbab, 2004, Dastani et al., 2005]

A Classical Example: Manifold [Arbab et al., 1993]

Main features

- coordinators
- control-driven evolution
  - events without parameters
- stateful communication
- coordination via topology
- fine-grained coordination
- typical example: sort-merge

Control-oriented Models: Impact on Design

Which abstractions?

- producer-consumer pattern
- point-to-point communication
- coordinator
- coordination as configuration of topology

Which systems?

- fine-grained abstractions
- fine-tuned control
- good for small-scale, closed systems

An Evolutionary Pattern?

Paradigms of sequential programming

- imperative programming with “goto”
- structured programming (procedure-oriented)
- object-oriented programming (data-oriented)
- agent-oriented programming (autonomy-oriented)

Paradigms of coordination programming

- message-passing coordination
- control-oriented coordination
- data-oriented coordination

? . . .

Data-oriented Models I

Communication channel

- shared memory abstraction
- stateful channel

Processes

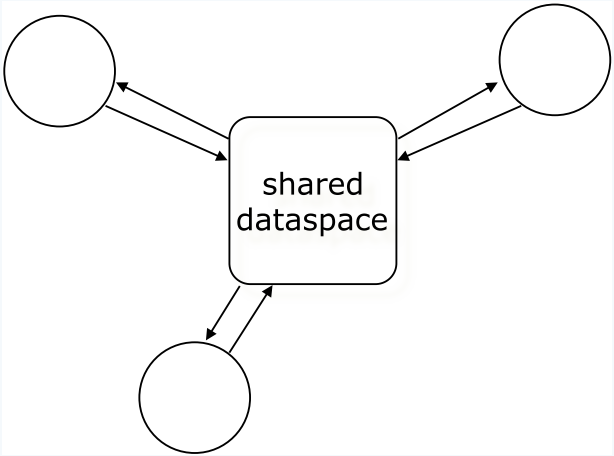
- emitting / receiving data / information

Coordination

- access / change / synchronise on shared data

Data-oriented Models II

Shared dataspace: constraint on communication



Data-oriented Models III

General features

- expressive communication abstraction

→ information-based design

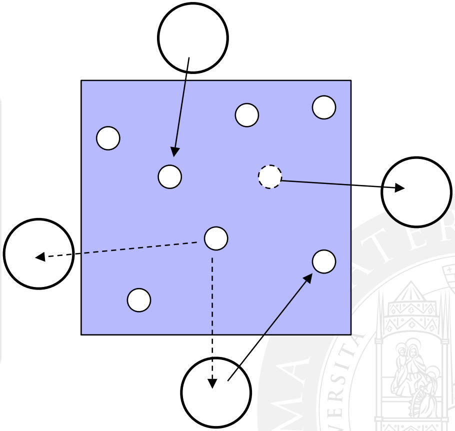
- possible spatio-temporal uncoupling
- does *no control* mean *no flexibility*?
- examples
  - Gamma / chemical coordination [Banǎtre et al., 2001]
  - LINDA & friends / tuple-based coordination [Rossi et al., 2001]

Next in Line. . .

The Tuple-space Meta-model

The basics

- *coordinables* synchronise, cooperate, compete
    - based on *tuples*
    - available in the *tuple space*
    - by *associatively* accessing, consuming and producing tuples



Tuple-based / Space-based Coordination Systems

Adopting the constructive coordination meta-model [Ciancarini, 1996]

coordination media *tuple spaces*

- as multiset / bag of data objects / structures called *tuples*

communication language *tuples*

- as ordered collections of (possibly heterogeneous) information items

coordination language *tuple space primitives*

- as a set of operations to put, browse and retrieve tuples to/from the space

Linda: The Communication Language [Gelernter, 1985]

Communication Language

tuples ordered collections of possibly heterogeneous information chunks

- examples: p(1), printer('HP',dpi(300)), [0,0.5], matrix(m0,3,3,0.5), tree_node(node00,value(13),left(_),right(node01)), ...

templates / anti-tuples specifications of set / classes of tuples

- examples: p(X), [?int,?int], tree_node(N), ...

tuple matching mechanism the mechanism that matches tuples and templates

- examples: pattern matching, *unification*, ...

Linda: The Coordination Language [Gelernter, 1985] I

`out(T)`

- `out(T)` puts tuple `T` into the tuple space

  examples `out(p(1))`, `out(0,0.5)`, `out(course('Antonio Natali','Poetry',hours(150))` ...

Linda: The Coordination Language [Gelernter, 1985] II

in(TT)

- `in(TT)` retrieves a tuple matching template `TT` from to the tuple space

  destructive reading the tuple retrieved is removed from the tuple centre

  non-determinism if more than one tuple matches the template, one is chosen non-deterministically

  suspensive semantics if no matching tuples are found in the tuple space, operation execution is suspended, and woken when a matching tuple is finally found

  examples `in(p(X))`, `in(0,0.5)`, `in(course('Antonio Natali',Title,hours(X))` ...

Linda: The Coordination Language [Gelernter, 1985] III

rd(TT)

- rd(TT) retrieves a tuple matching template TT from to the tuple space

  non-destructive reading the tuple retrieved is left untouched in the tuple centre

  non-determinism if more than one tuple matches the template, one is chosen non-deterministically

  suspensive semantics if no matching tuples are found in the tuple space, operation execution is suspended, and awakened when a matching tuple is finally found

  examples `rd(p(X))`, `rd(0,0.5)`, `rd(course('Alessandro Ricci','Operating Systems',hours(X))` ...

Linda Extensions: Predicative Primitives

`inp(TT)`, `rdp(TT)`

- both `inp(TT)` and `rdp(TT)` retrieve tuple `T` matching template `TT` from the tuple space

  = `in(TT)`, `rd(TT)` (non-)destructive reading, non-determinism, and syntax structure is maintained

  ≠ `in(TT)`, `rd(TT)` suspensive semantics is lost: this *predicative* versions primitives just fail when no tuple matching `TT` is found in the tuple space

  success / failure predicative primitives introduce *success / failure semantics*: when a matching tuple is found, it is returned with a success result; when it is not, a failure is reported

Linda Extensions: Bulk Primitives I

`in_all(TT), rd_all(TT)`

- Linda primitives deal with one tuple at a time
  - some coordination problems require more than one tuple to be handled by a single primitive
- `rd_all(TT)`, `in_all(TT)` get all tuples in the tuple space matching with `TT`, and returns them all
  - no suspensive semantics: if no matching tuple is found, an empty collection is returned
  - no success / failure semantics: a collection of tuple is always successfully returned—possibly, an empty one
  - in case of logic-based primitives / tuples, the form of the primitive are `rd_all(TT,LT)`, `in_all(TT,LT)` (or equivalent), where the (possibly empty) list of tuples unifying with `TT` is unified with `LT`
  - (non-)destructive reading: `in_all(TT)` consumes all matching tuples in the tuple space; `rd_all(TT)` leaves the tuple space untouched

Linda Extensions: Bulk Primitives II

Other bulk primitives

- many other bulk primitives have been proposed and implemented to address particular classes of problems
- most of them too specific to be considered as a general extension to Linda, and for inclusion in tuple-based models in general

LINDA Extensions: Multiple Tuple Spaces

`ts ? out(T)`

- LINDA tuple space might be a bottleneck for coordination
- many extensions have focussed on making a multiplicity of tuple spaces available to processes
  - each of them encapsulating a portion of the coordination load
  - either hosted by a single machine, or distributed across the network
- syntax required, and dependent on particular models and implementations
  - a space for tuple space names, possibly including network location
  - operators to associate LINDA operators to tuple spaces
- for instance, `ts @ node ? out(p)` may denote the invocation of operation `out(p)` over tuple space `ts` on node `node`

Main Features of Tuple-based Coordination

Main features of the LINDA model

tuples
: a tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort

generative communication
: until explicitly withdrawn, the tuples generated by coordinables have an independent existence in the tuple space; a tuple is equally accessible to all the coordinables, but is bound to none

associative access
: tuples in the tuple space are accessed through their content & structure, rather than by name, address, or location

suspensive semantics
: operations may be suspended based on unavailability of matching tuples, and be woken up when such tuples become available

Features of Linda: Tuples

tuple an ordered collection of knowledge chunks, possibly heterogeneous in sort

- a record-like structure
- with no need of field names
- easy aggregation of knowledge
- raw semantic interpretation: a tuple contains all information concerning an given item

tuple structure based on

- arity
- type
- position
- information content

tuple templates / anti-tuples

- to describe / define sets of tuples

matching mechanism

- to define belongingness to a set

Features of Linda: Generative Communication

Communication orthogonality

- both senders and the receivers can interact even without having prior knowledge about each others

space uncoupling	no need to coexist in space for two processes to interact
time uncoupling	no need for simultaneity for two processes to interact
name uncoupling	no need for names for processes to interact

Features of Linda: Associative Access

Content-based coordination

synchronisation based on tuple *content* & *structure*

- absence / presence of tuples with some content / structure determines the overall behaviour of the coordinables, and of the coordinated system in the overall
- based on tuple templates & matching mechanism

information-driven coordination

- patterns of coordination based on data / information availability
- based on tuple templates & matching mechanism

reification

- making events become tuples
- grouping classes of events with tuple syntax, and accessing them via tuple templates

Features of Linda: Suspensive Semantics

Blocking primitives

- `in` & `rd` primitives in LINDA have a suspensive semantics
  - the coordination medium makes the primitives waiting in case a matching tuple is not found, and wakes it up when such a tuple is found
  - the coordinable invoking the suspensive primitive is expected to wait for its successful completion
- twofold wait

  in the coordination medium the operation is first (possibly) suspended, then (possibly) served: coordination based on absence / presence of tuples belonging to a given set

  in the coordination entity the invocation may cause a wait-state in the invoker: hypothesis on the internal behaviour of the coordinable

Our Running Example: The Dining Philosophers Problem

Dining Philosophers [Dijkstra, 2002]

- in the classical Dining Philosopher problem, *N* philosophers *share N chopsticks* and a spaghetti bowl
- each philosopher either *eats* or *thinks*
- each philosopher needs a *pair of chopsticks* to eat—and can access the two chopsticks on his left and on his right
- each chopstick is *shared* by two adjacent philosophers
- when a philosopher needs to *think*, he *gets rid* of chopsticks

Concurrency issues in the Dining Philosophers Problem

shared resources	two adjacent philosophers cannot eat simultaneously
starvation	if one philosopher eats all the time, the two adjacent philosophers will starve
deadlock	if every philosopher picks up the same (say, the left) chopstick at the same time, all of them may wait indefinitely for the other (say, the right) chopstick so as to eat
fairness	if a philosopher releases one chopstick before the other one, it favours one of his adjacent philosophers over the other one

Dining Philosophers in Linda

- the spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by the tuple space
- chopsticks are represented as tuples `chop(`*i*`)`, that represents the left chopstick for the $i - th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i + 1) modN$ (right)
- philosophers try to eat by getting their chopstick pairs from the tuple space as a pair of tuples `chop(`*i*`) chop(`*i+1 mod N*`)`
- philosophers start to think by releasing their own chopstick pairs to the tuple space as a pair of tuples `chop(`*i*`) chop(`*i+1 mod N*`)`

! *in the following, we will use Prolog for philosopher agents*

Dining Philos in Linda: A Simple Philosopher Protocol

Philosopher using `ins` and `outs`

Dining Philos in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-




    !, philosopher(I,J).
```

Dining Philos in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-




    !, philosopher(I,J).
```

Dining Philos in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                                  % thinking



    !, philosopher(I,J).
```

Dining Philos in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)), in(chop(J)),           % waiting to eat



    !, philosopher(I,J).
```

Dining Philos in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                           % thinking
    in(chop(I)), in(chop(J)),        % waiting to eat


    !, philosopher(I,J).
```

Dining Philos in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)), in(chop(J)),           % waiting to eat
    eat,                                % eating

    !, philosopher(I,J).
```

Dining Philos in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                                  % thinking
    in(chop(I)), in(chop(J)),               % waiting to eat
    eat,                                    % eating
    out(chop(I)), out(chop(J)),             % waiting to think
    !, philosopher(I,J).
```

Dining Philos in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)), in(chop(J)),           % waiting to eat
    eat,                                % eating
    out(chop(I)), out(chop(J)),         % waiting to think
    !, philosopher(I,J).
```

Dining Philos in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)), in(chop(J)),           % waiting to eat
    eat,                                % eating
    out(chop(I)), out(chop(J)),         % waiting to think
    !, philosopher(I,J).
```

Issues

Dining Philos in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)), in(chop(J)),           % waiting to eat
    eat,                                % eating
    out(chop(I)), out(chop(J)),         % waiting to think
    !, philosopher(I,J).
```

Issues

+ shared resources handled correctly

Dining Philos in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)), in(chop(J)),           % waiting to eat
    eat,                                % eating
    out(chop(I)), out(chop(J)),         % waiting to think
    !, philosopher(I,J).
```

Issues

+ shared resources handled correctly
– starvation, deadlock and unfairness still possible

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using `ins`, `inps` and `outs`

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using `ins`, `inps` and `outs`

```
philosopher(I,J) :-
```

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-









    !, philosopher(I,J).
```

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                            % thinking
    in(chop(I)),                      % waiting to eat
    (  inp(chop(J)),                  % if other chop available
       eat,                           % eating
       out(chop(I)), out(chop(J)),    % waiting to think
       ;                              % otherwise
       out(chop(I))                   % releasing unused chop
    )
    !, philosopher(I,J).
```

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (  inp(chop(J)),                    % if other chop available
       eat,                             % eating
       out(chop(I)), out(chop(J)),      % waiting to think
       ;                                % otherwise
       out(chop(I))                     % releasing unused chop
    )
    !, philosopher(I,J).
```

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (  inp(chop(J)),                    % if other chop available
       eat,                             % eating
       out(chop(I)), out(chop(J)),      % waiting to think
       ;                                % otherwise
       out(chop(I))                     % releasing unused chop
    )
    !, philosopher(I,J).
```

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                             % thinking
    in(chop(I)),                       % waiting to eat
    (  inp(chop(J)),                   % if other chop available
       eat,                            % eating
       out(chop(I)), out(chop(J)),     % waiting to think
       ;                               % otherwise
       out(chop(I))                    % releasing unused chop
    )
    !, philosopher(I,J).
```

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                               % thinking
    in(chop(I)),                         % waiting to eat
    (   inp(chop(J)),                    % if other chop available
        eat,                             % eating
        out(chop(I)), out(chop(J)),      % waiting to think
        ;                                % otherwise
        out(chop(I))                     % releasing unused chop
    )
    !, philosopher(I,J).
```

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                  % thinking
    in(chop(I)),                            % waiting to eat
    (  inp(chop(J)),                        % if other chop available
       eat,                                 % eating
       out(chop(I)), out(chop(J)),          % waiting to think
       ;                                    % otherwise
       out(chop(I))                         % releasing unused chop
    )
    !, philosopher(I,J).
```

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                           % thinking
    in(chop(I)),                     % waiting to eat
    (   inp(chop(J)),                % if other chop available
        eat,                         % eating
        out(chop(I)), out(chop(J)), % waiting to think
        ;                            % otherwise
        out(chop(I))                 % releasing unused chop
    )
    !, philosopher(I,J).
```

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                 % thinking
    in(chop(I)),                           % waiting to eat
    (   inp(chop(J)),                      % if other chop available
       eat,                                % eating
       out(chop(I)), out(chop(J)),         % waiting to think
       ;                                   % otherwise
       out(chop(I))                        % releasing unused chop
    )
    !, philosopher(I,J).
```

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                 % thinking
    in(chop(I)),                           % waiting to eat
    (   inp(chop(J)),                      % if other chop available
        eat,                               % eating
        out(chop(I)), out(chop(J)),        % waiting to think
        ;                                  % otherwise
        out(chop(I))                       % releasing unused chop
    )
    !, philosopher(I,J).
```

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                    % thinking
    in(chop(I)),                              % waiting to eat
    (   inp(chop(J)),                         % if other chop available
       eat,                                   % eating
       out(chop(I)), out(chop(J)),            % waiting to think
       ;                                      % otherwise
       out(chop(I))                           % releasing unused chop
    )
    !, philosopher(I,J).
```

Issues

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                  % thinking
    in(chop(I)),                            % waiting to eat
    (   inp(chop(J)),                       % if other chop available
        eat,                                % eating
        out(chop(I)), out(chop(J)),         % waiting to think
        ;                                   % otherwise
        out(chop(I))                        % releasing unused chop
    )
    !, philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                               % thinking
    in(chop(I)),                         % waiting to eat
    (   inp(chop(J)),                    % if other chop available
        eat,                             % eating
        out(chop(I)), out(chop(J)),      % waiting to think
        ;                                % otherwise
        out(chop(I))                     % releasing unused chop
    )
    !, philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
− starvation and unfairness still possible

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                               % thinking
    in(chop(I)),                         % waiting to eat
    (  inp(chop(J)),                     % if other chop available
       eat,                              % eating
       out(chop(I)), out(chop(J)),       % waiting to think
       ;                                 % otherwise
       out(chop(I))                      % releasing unused chop
    )
    !, philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                % thinking
    in(chop(I)),                          % waiting to eat
    (   inp(chop(J)),                     % if other chop available
        eat,                              % eating
        out(chop(I)), out(chop(J)),       % waiting to think
        ;                                 % otherwise
        out(chop(I))                      % releasing unused chop
    )
    !, philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
− starvation and unfairness still possible
− not-so-trivial philosopher's interaction protocol
    - part of the coordination load is on the coordinables

Dining Philos in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                               % thinking
    in(chop(I)),                         % waiting to eat
    (  inp(chop(J)),                     % if other chop available
       eat,                              % eating
       out(chop(I)), out(chop(J)),       % waiting to think
       ;                                 % otherwise
       out(chop(I))                      % releasing unused chop
    )
    !, philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philos in Linda: Yet Another Philosopher Protocol

Philosopher using `ins` and `outs` with chopstick pairs `chops(I,J)`

Dining Philos in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
```

Dining Philos in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-





    !, philosopher(I,J).
```

Dining Philos in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
 think,                         % thinking
    in(chops(I,J)),             % waiting to eat
    eat,                        % eating
    out(chops(I,J)),            % waiting to think
    !, philosopher(I,J).
```

Dining Philos in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
 think,                          % thinking
    in(chops(I,J)),              % waiting to eat
    eat,                         % eating
    out(chops(I,J)),             % waiting to think
    !, philosopher(I,J).
```

Dining Philos in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
 think,                        % thinking
    in(chops(I,J)),            % waiting to eat
    eat,                       % eating
    out(chops(I,J)),           % waiting to think
    !, philosopher(I,J).
```

Dining Philos in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
 think,                         % thinking
    in(chops(I,J)),             % waiting to eat
    eat,                        % eating
    out(chops(I,J)),            % waiting to think
    !, philosopher(I,J).
```

Dining Philos in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
 think,                          % thinking
    in(chops(I,J)),              % waiting to eat
    eat,                         % eating
    out(chops(I,J)),             % waiting to think
    !, philosopher(I,J).
```

Issues

Dining Philos in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
 think,                         % thinking
    in(chops(I,J)),             % waiting to eat
    eat,                        % eating
    out(chops(I,J)),            % waiting to think
    !, philosopher(I,J).
```

Issues

+ fairness, no deadlock

Dining Philos in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
 think,                         % thinking
    in(chops(I,J)),             % waiting to eat
    eat,                        % eating
    out(chops(I,J)),            % waiting to think
    !, philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol

Dining Philos in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
 think,                        % thinking
    in(chops(I,J)),            % waiting to eat
    eat,                       % eating
    out(chops(I,J)),           % waiting to think
    !, philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
– shared resources not handled properly

Dining Philos in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
 think,                          % thinking
    in(chops(I,J)),              % waiting to eat
    eat,                         % eating
    out(chops(I,J)),             % waiting to think
    !, philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
– shared resources not handled properly
– starvation still possible

Dining Philosophers in Linda: Where is the Problem?

- coordination is limited to writing, reading, consuming, suspending on one tuple at a time
    - the behaviour of the coordination medium is fixed once and for all
    - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- bulk primitives are not a general-purpose solution
    - adding ad hoc primitives does not solve the problem in general
    - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- as a result, the coordination load is typically charged upon coordination entities
    - this does not fit open scenarios
    - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Tuple-based Models: Solution?

- making the behaviour of the coordination medium *adjustable* according to the coordination problem
  - if the behaviour of the coordination medium is *not* be fixed once and for all, and can be defined in accordance to the coordination needs
  - then, in principle all coordination problems may fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- in this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of *coordination policies*
  - enacted in terms of *coordinative behaviour* of the coordination media
- what is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Summing Up

Coordination for distributed system engineering

- engineering the space of interaction among components

Coordination as governing interaction

- enabling vs. governing

Classes and features of coordination models

- control-oriented vs. data-oriented models
- tuple-based models
  - features
  - issues

References I

Arbab, F. (2004).
Reo: A channel-based coordination model for component composition.
Mathematical Structures in Computer Science, 14:329–366.

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

Banătre, J.-P., Fradet, P., and Le Métayer, D. (2001).
Gamma and the chemical reaction model: Fifteen years after.
In Calude, C. S., Păun, G., Rozenberg, G., and Salomaa, A., editors, *Multiset Processing. Mathematical, Computer Science, and Molecular Computing Points of View*, volume 2235 of *LNCS*, pages 17–44. Springer.

Brooks, R. A. (1991).
Intelligence without reason.
In Mylopoulos, J. and Reiter, R., editors, *12th International Joint Conference on Artificial Intelligence (IJCAI 1991)*, volume 1, pages 569–595, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

References II

Ciancarini, P. (1996).
Coordination models and languages as software integrators.
ACM Computing Surveys, 28(2):300–302.

Dastani, M., Arbab, F., and de Boer, F. S. (2005).
Coordination and composition in multi-agent systems.
In Dignum, F., Dignum, V., Koenig, S., Kraus, S., Singh, M. P., and Wooldridge, M. J., editors, *4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 439–446, Utrecht, The Netherlands. ACM.

Deugo, D., Weiss, M., and Kendall, E. (2001).
Reusable patterns for agent coordination.
In [Omicini et al., 2001], chapter 14, pages 347–368.

Dijkstra, E. W. (2002).
Co-operating sequential processes.
In Hansen, P. B., editor, *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, chapter 2, pages 65–138. Springer.
Reprinted. 1st edition: 1965.

References III

Erl, T. (2005).
Service-Oriented Architecture: Concepts, Technology, and Design.
Prentice Hall / Pearson Education International, Upper Saddle River, NJ, USA.

FIPA ACL (2002).
Agent Communication Language Specifications.
Foundation for Intelligent Physical Agents (FIPA).

Fredriksson, M. and Gustavsson, R. (2004).
Online engineering and open computational systems.
In Bergenti, F., Gleizes, M.-P., and Zambonelli, F., editors, *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organization*, pages 377–388. Kluwer Academic Publishers.

Gelernter, D. (1985).
Generative communication in Linda.
ACM Transactions on Programming Languages and Systems, 7(1):80–112.

Gelernter, D. and Carriero, N. (1992).
Coordination languages and their significance.
Communications of the ACM, 35(2):97–107.

References IV

Goldin, D. Q., Smolka, S. A., and Wegner, P., editors (2006a).
Interactive Computation: The New Paradigm.
Springer.

Goldin, D. Q., Smolka, S. A., and Wegner, P., editors (2006b).
Interactive Computation: The New Paradigm.
Springer Berlin Heidelberg.

Holzbacher, A.-A. (1996).
A software environment for concurrent coordinated programming.
In Ciancarini, P. and Hankin, C., editors, *Coordination Languages and Models*, volume 1061 of *LNCS*, pages 249–266. Springer-Verlag.
1st International Conference (COORDINATION '96) Cesena, Italy, April 15–17, 1996.

Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D., and Mann, W. (1995).
Specification and analysis of system architecture using Rapide.
IEEE Transactions on Software Engineering, 21(4):336–354.

Milner, R. (1993).
Elements of interaction: Turing award lecture.
Communications of the ACM, 36(1):78–89.

References V

Omicini, A., Zambonelli, F., Klusch, M., and Tolksdorf, R., editors (2001).
Coordination of Internet Agents: Models, Technologies, and Applications.
Springer Berlin Heidelberg.

Papadopoulos, G. A. and Arbab, F. (1998).
Coordination models and languages.
In Zelkowitz, M. V., editor, *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 329–400. Academic Press.

Rossi, D., Cabri, G., and Denti, E. (2001).
Tuple-based technologies for coordination.
In [Omicini et al., 2001], chapter 4, pages 83–109.

Wegner, P. (1997).
Why interaction is more powerful than algorithms.
Communications of the ACM, 40(5):80–91.

Wegner, P. and Goldin, D. (1999).
Mathematical models of interactive computing.
Technical report, Brown University, Providence, RI, USA.

Coordination of Distributed Systems

Distributed Systems / Paradigms
Sistemi Distribuiti / Paradigmi

Andrea Omicini
`andrea.omicini@unibo.it`

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2017/2018