Goals & Issues of Distributed Systems

Distributed Systems
Sistemi Distribuiti

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
ALMA MATER STUDIORUM – Università di Bologna a Cesena

Academic Year 2016/2017

About these Slides

- the following slides borrow a great deal from the slides coming with the book adopted as the basic one for this course
  [Tanenbaum and van Steen, 2007]
- material from those slides (including pictures) has been re-used in the following, and integrated with new material according to the personal view of the professor
- every problem or mistake contained in these slides, however, should be attributed to the sole responsibility of this course's professor
- the goal for students, here, is to be exposed to the classical *general view* on distributed systems that the technical and scientific community has developed in the last decades

Next in Line. . .

Distributed System: Classic Definitions I

A distributed system can be defined as. . .

. . . a collection of *independent* computers that *appears* to its users as a *single coherent system* [Tanenbaum and van Steen, 2007]

User's view

- a possible definition, accounting for an *observational*, user-oriented view
- we may also call it the computer scientist definition of a distributed system
- from an engineering viewpoint, it is a sort of *a posteriori* definition

Distributed System: Classic Definitions II

A distributed system can be defined as...

... a collection of *autonomous* computational entities *conceived* as a *single coherent system* by its designer

Engineer's view

- another possible definition, accounting for a *constructive*, design-oriented view
- we may also call it the computer engineer definition of a distributed system
- from an engineering viewpoint, it is a sort of *a priori* definition

Distributed System: Classic Definitions III

Remarks

- a distributed system is made of a multiplicity of *components*
  - independent / autonomous computational entities (computers, microprocessors, . . .)
  - ! no definition focus specifically on either computational processes or computing devices, here
  - *no assumptions* on their individual nature, structure, behaviour, . . .
  - heterogeneity
- a distributed system can be seen as a single coherent system
  - according to either the user's view or the engineer's view—or both
  - need for *coherence* over multiplicity and heterogeneity

Distributed System: Another Definition

A distributed system can be defined as...

... one in which components located at *networked* computers *communicate* and *coordinate* their actions only by passing *messages*.

[Coulouris et al., 2012]

Remarks

This definition focusses on the following features of distributed systems:

- physical *distribution* of components
- the role of *interaction*—network-based communication and coordination
- uncoupling in terms of *control*

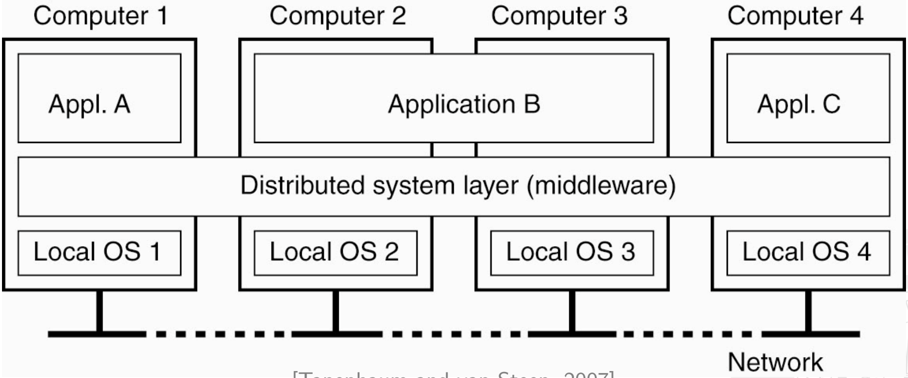
Distributed System: Working as One, Looking as One

Issues: coherence & uniformity

collaboration
: in order to achieve coherence, many *autonomous* entities should *collaborate*—that is, work altogether as a single (coherent) system

amalgamation
: in order to achieve uniformity, many *heterogeneous* entities should *amalgamate*—that is, look altogether as a single (uniform) system

An Architectural View of Distributed Systems: Middleware



[Tanenbaum and van Steen, 2007]

- a distributed system organised as middleware
- the *middleware layer* extends over multiple machines, and offers each application the *same interface*

An Architectural View of Distributed Systems: Old vs. New

Moving from a view of non-distributed computational systems. . .

- . . . by trying to extend the same old interpretation of systems
- good for preserving good habits
- bad when looking for new ideas and new problems

! *middleware* has obviously a role to play here

Middleware: A Principled Solution

Middleware for collaboration & amalgamation

Through *separation*, the middleware layer. . .

- . . . enables *meaningful interaction* between autonomous distributed components
  - communication issues like the language for messages and its semantic interpretation
- . . . hides differences in technology, structure, behaviour
  - providing both applications and components with a common shared interface—in the same way as operating systems

Next in Line. . .

What Made Computational Systems *Distributed*?

At the very beginning

- computers were huge & expensive machines
- computers were *islands*
  - computer science – at the time, as the art of computer programming – was born upon such machines

. . . then, drastic advances in Electronics and TLC occurred

- microprocessor technology made computational *devices* more and more powerful and cheap
- high-speed computer networks made *interconnection* of computational entities possible at a wide range of scales and speeds
  - the *scope* and *goal* of computer science changed dramatically

↓ from centralised (single-processor) systems

→ to decentralised (multi-processor), distributed systems

Why Should We Build a Distributed System?

A distributed system is *easy to build*

- hardware, software, and network components are easy to find & use
  - and to be *put together* somehow
- however, at a first sight, distribution apparently introduces *problems*, rather than solving them
  - ? why should we build a system as a distributed one?
  - ! in particularly, given that it is not easy to make a distributed system *actually work*

Making Distributed System Worth the Effort

Four goals for a distributed system...

- making (distributed, remote) *resources* available for use
- allowing *distribution* of resources to be hidden whenever unnecessary
- promoting openness
- promoting scalability

... plus, a fifth goal

- promoting situatedness

Making Resources Available

Resources are physically distributed

- a good reason to build a distributed system is to make them distributed resources *available* as they would belong to a single system

What is a resource?

Anything that ...

- ... could be in any way connected to a computational system
- ... anyone could legitimately use

E.g., printers, scanners, storage devices, distributed sensors, ...

By making interaction possible between users and resources, distributed systems are *enablers* of sharing, information exchange, collaboration, ...

Distribution Transparency

Physical distribution is not a feature, sometimes

- a(nother) good reason to build a distributed system is to make physical distribution *irrelevant* from the user's viewpoint

Transparency

- hiding non-relevant properties of the system's components and structure is called *transparency*
- there exists a number of different and useful sorts of transparency, according to the property hidden to the user's perception

By hiding non-relevant properties to users, distributed systems provide users with a *higher level of abstraction*

Types of Transparency in a Distributed System

- **access** hide differences in data *representation* and in the *access* to resources
- **location** hide the *location* of a resource
- **migration** hide the *change of location* of a resource
- **relocation** hide the *motion* of a resource
- **replication** hide that a resource is *replicated*
- **concurrency** hide the *sharing* of a resource by multiple users
- **failure** hide the *failure* and subsequent *recovery* of a resource

Access Transparency

Heterogeneity in representation and use

- different *data representation*
- different *component structure*
- different *resource* usage *interface* / *protocols*

All of them should be hidden from the user's view, whenever possible & non-relevant

Providing a homogeneous view over heterogeneity

- a distributed system should hide heterogeneity, by providing uniform/homogeneous access to data, components, resources

Location Transparency

Location of users and resources

- often, the physical location of a resource is not relevant for its use by the users—and, viceversa, the location of users is often irrelevant for the resource
    - e.g., the position of a storage facility, or of a single printer in a cluster of printers in a lab

Hiding physical distribution of users and resources

- distributed systems should hide physical distribution, whenever possible & non-relevant

Naming is essential

- there should exist a system of logical *identifiers*, not bound to physical location
    - e.g., URLs

Migration Transparency

Resources might be *mobile*

- locations change within a distributed system
- which has to maintain its coherence anyway

A distributed system should allow *migration* of resources

- without losing coherence
- without losing functionality

Also *users* might move

- some years ago this was a sort of secondary aspect
- nowadays, user's mobility is typically a core issue in the modelling and engineering of artificial systems

Relocation Transparency

Resources should be still accessible when moving

migration
: should not prevent users to access resources, *while they are changing their location*

relocation
: transparency is in some sense a specialised, *on line* version of migration transparency

A distributed system could be useful to *allow access* to *mobile resources* by *mobile users*, by hiding changes in location (migration transparency), even while changes are actually occurring (relocation transparency)

Replication Transparency I

Sometimes *replication* helps

Like, for instance,

- in providing a *local copy* of a *resource* to local agents/users—so as to
    - reduce bandwidth consumption
    - provide faster access
- in promoting *tolerance to failures*
    - through *redundancy*

Replication Transparency II

Whatever the motivation behind replication ...

... replication is not something a user should worry about

- all replicas should be accessible in the same, *transparent* way
- so they should have the *same name*
- and should be essentially in the *same state*, so to be *apparently one* and the same thing for each and every user

A distributed system could exploit *replication* techniques for many reasons, but should be able at the same time to make it *invisible* to the users

Concurrency Transparency I

Activity in a distributed systems involves *independent* entities

- users and resources are *distributed*, and work *autonomously*, in a *concurrent* way
- for instance, two users may try to exploit the same resource at the same time
- typically, no user need to be bothered with these facts—like, "another user is accessing the same database you are accessing just now", who cares?

Concurrency Transparency II

Concurrency in activities should be *hidden* to users

- while shared access to resources could be done cooperatively, it is often the case that users should access competitively to resources
- a distributed system could take care of this, by defining access policies governing concurrent sharing of resources
- possibly, transparently to the users

Concurrency & Consistency

The problem of *consistency*

- when many users access the same resource concurrently, *consistency* of its state is in jeopardy—but should be ensured anyway
- a distributed system should take care of ensuring resource safety even under concurrent accesses
- as usual, transparently to the users

A distributed system should take care of allowing transparent concurrent access to resources, while ensuring consistency of resources

Failure in Distributed Systems

Failure in a distributed system is any failure *anywhere* in the system

- "*You know you have [a distributed system] when the crash of a computer you've never heard of stops you from getting any work done.*" (L. Lamport)
- distribution might be either a source of problems or a blessing
- it means that a failure could occur anywhere, but also that a part of the system is likely keep on working
  - *distributed failure* is hard to control
  - *partial failure* is possible, and much better than *total failure* of centralised systems

Distribution should work as a *feature*

Failure Transparency

What does this mean?

- masking failures under realistic assumptions
- hiding failure of resources to users

Being failure transparent is a hard problem

- e.g., the problem of *latency*
  - how do we distinguish between a dead resource and a very slow one?
  - is “silence” from a resource originated by slowness, deliberate choice, resource failure, or network failure?

A distributed system should exploit distribution to *reduce* the impact of *partial failure* onto the overall system, *hiding failures* to *users* as much and often as possible.

Degree of Transparency in a Distributed System I

Hiding distribution is not always the best idea

- for instance, users may move and be subject to different time zones—this could lead to funny situations, if hidden, such as a download ending before it started
- also, you may appreciate to know where the file server is located (US, Japan, Europe?) when choosing from where you will download the new Ultra HD Pokémon movie from home
  - unless it is replicated in such a way that it does not matter

Degree of Transparency in a Distributed System II

Trade-off between transparency and information

- it typically concerns performance—yet, it is not limited to this
- location-awareness is often a feature
- every engineer should find out the precise *degree of transparency* its distributed system should feature, by taking into account other issues like performance, understandability, . . .

Next in Line. . .

Openness

What is openness?

- essentially, the property of a system to work with a number and sort of components that is not set once and for all at design time

$\leftarrow$ open systems are typically *designed to be open*

$\rightarrow$ open systems are fundamentally *unpredictable*

Designing over unpredictability requires predictable items

- something needs to be *a priori* shared between the system and the (potential) components
- like, e.g., standard rules for services syntax and semantics, message interchange, ...

Interfaces for Open Systems

Interfaces to specify syntax for accessing services

- IDL (Interface Definition Languages) to define how *interfaces* are specified
- they capture syntax, rather than semantics—often, they do not specify the *protocol*, too

Issues for Open Systems

Interoperability

interoperability measures how easy / difficult is to make one component / system work with different ones based on some standard-based specifications

portability measures how much an application (or, a portion of it) can be moved to a different distributed system and keep working

extensibility measures how easy / difficult is to add new components and functionality to an existing distributed system

Separating Policies from Mechanisms I

Openness mandates for a clean architecture

- external interfaces are not enough
- components should be small and focussed enough to be easily modified / replaced
- internal specifications should be as neat as the external ones

Separating Policies from Mechanisms II

Components providing mechanisms should not impose policies

- mechanisms are the basic *operational bricks* made available to designers and developers
  - mechanisms should be *neutral*—which means they should not force any particular *choice*, letting systems open to different policy specifications
- policies exploit mechanisms to implement *choices* in a system, ruling admissible behaviour of components as well as their mutual interactions
  - policies should be either *encapsulated* into other components or *delegated* to users
  - (assuming the two things are really different)
- clear separation between mechanisms and policies should be enforced

Scalability

World-wide scale changes everything

- often, few realistic assumptions can be done on the actual "size" of a distributed system at design time
- there, *size* might mean actual size in number of components, but also in geographical distribution

Dimensions of scalability [Neuman, 1994]

- a system scales up when the *number* of users and resources grows
- a system scales up when the *geographical distribution* of users and resources extends
- a system scales up when it spans over a growing number of distinct *administrative domains*

Scalability is an issue whenever any dimension of a distributed system changes its order of magnitude

Scalability Problems: Scaling with Respect to Size

Examples of scalability limitations [Tanenbaum and van Steen, 2007]

centralised services a single server for all users

centralised data a single database for all distributed components

centralised algorithms complete information is assumed to be available in one single place

Centralisation

Making things centralised might be necessary

- even though a single server is a bottleneck, it could be a necessity in case of security problems, or, normative requirements
- even though a single collection of data is a bottleneck, it could still be needed if replication is insecure
- sometimes, the most efficient algorithm from a theoretical viewpoint might be a centralised one

However, *centralisation hinder scalability*, and should be avoided in general in distributed systems whenever possible

Decentralised vs. Centralised Algorithms I

The trouble with centralised algorithms [Raynal, 2013]

- data should flow from the whole network to and from the place where the centralised algorithm works
- the network would be overloaded
- any transmission problem would cause problems to the overall algorithm

→ only decentralised algorithms should be used in distributed systems

Decentralised vs. Centralised Algorithms II

Characteristics of decentralised algorithms [Kshemkalyani and Singhal, 2011]

- no machine has complete information about the system state
- machines make decisions based only on local information
- failure of one machine does not ruin the algorithm
- there is no implicit assumption that a global clock exists

Scalability Problems: Geographical Scalability I

The trouble with communication

- communication in LAN is typically synchronous—this does not scale up to WAN: e.g., how should timeouts be set up?
- communication in WAN is typically unreliable, and typically point-to-point—LAN broadcasting no longer an option: e.g., how could a service be located?

Shared troubles with size scalability

Centralisation is still a mess, since

- it requires computation and interaction capabilities to scale up with size
- it also leads to a *single point of failure*

Scalability Problems: Geographical Scalability II

Administration / organisation troubles

- e.g., within a single domain, users and components might be trusted: however, trust does not cross domain boundaries
- distributed systems typically extend over *multiple* administration / organisation *domains*
  - security measures are needed that may hinder scalability
  - policies may conflict

Techniques for Geographical Scalability

Three Basic Techniques [Neuman, 1994]

- hiding communication latency
  - asynchronous communication
- distribution
- replication

Scaling Techniques: Hiding Communication Latency I

The basic idea

Try to avoid wasting time waiting for remote responses to service requests whenever possible

Asynchronous communication

This basically means using *asynchronous communication* for requests whenever possible

- a request is sent by the application
- the application does not stop waiting for a response
- when a response come in, the application is interrupted and a handler is called to complete the request

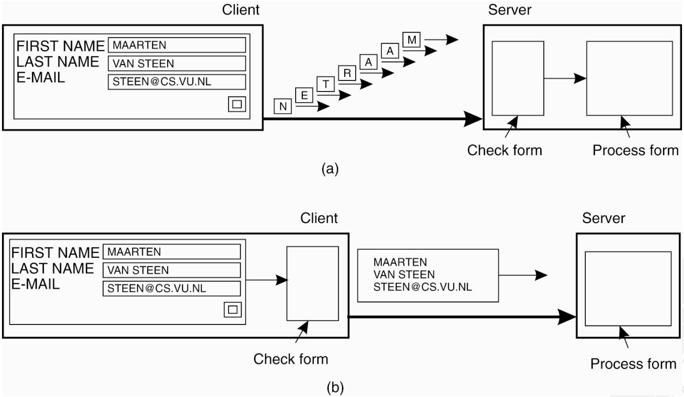
Scaling Techniques: Hiding Communication Latency II

Problem

Sometimes, asynchronous communication is not feasible

- like in Web application when a user is just waiting for the response
- alternative techniques like shipping code are needed—e.g., Javascript or Java Applets

Scaling Techniques: Code Shipping Example



The difference between letting (a) a server or (b) a client check forms as they are being filled [Tanenbaum and van Steen, 2007]

Scaling Techniques: Distribution

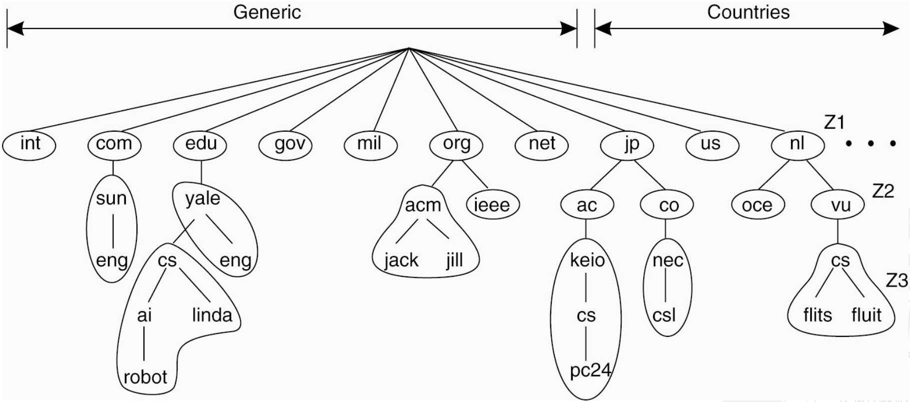
The basic idea

Taking a component, splitting it into parts, and spreading the parts across the system

Example: The Domain Name System (DNS)

- the DNS is hierarchically organised into a tree of *domains*
- domains are divided in non-overlapping *zones*
- the names in each zone are in charge of a single server
- e.g., `apice.unibo.it`
- the *naming service* is thus distributed across several machines, without centralisation

Scaling Techniques: Distribution Example



An example of dividing the DNS space into zones

[Tanenbaum and van Steen, 2007]

Scaling Techniques: Replication

The basic idea

- when degradation of performance occurs, *replicating* components across a distributed system may increase availability and solve problems of latency
- replication typically involves making a copy of a resource form the original location to a location in the proximity of the (potential) users

Caching

- is a special form of replication
- caching is making a copy of a resource, like replication
- however, caching is a decision by the client of a resource, replication by the owner of a resource

The Problem of Consistency

Duplicating a resource introduces *consistency* problems

- involving both caching and replication
- inconsistency is technically unavoidable, whenever copying a resource in a distributed setting
- the point is *how much inconsistency* could a system tolerate, and how it could be hidden from users and components of a distributed system

Scalability Problems: Administrative Scalability

The trouble with organisation

- maybe the most difficult problem: many non-technical problems to be solved, such as policy of organisation and human collaboration

A successful approach: Ignoring administrative domains

- users take over control: peer-to-peer technologies
- only a partial solution, nevertheless, something should be done

A recommendation for computer scientists & engineers

- not every organisation problem has a technical solution
- sometimes, personal interactions within an organisation are the only way to overcame a pseudo-technical problem

Situatedness I

What is situatedness? [Suchman, 1987]

- situatedness is in short the property of systems of being *immersed* in their environment
- that is, of being capable to (timely) *perceive* and *produce* environment change, by suitably dealing with environment events
- mobile, adaptive, and pervasive computing systems have emphasised the key role of situatedness for nowadays computational systems [Zambonelli et al., 2011]

Situatedness II

Situatedness & context-awareness

- computational systems are more and more affected by their *physical nature*
- this is not due to a lack of abstraction: indeed, we are suitably layering systems – including hardware and software layers –, where separation between the different layers is clear and well defined
- instead, this is mostly due to the increasingly *complex requirements* for computational systems, which mandate for an ever-increasing context-awareness [Baldauf et al., 2007]
- defining the notion of *context* is a complex matter [Omicini, 2002], with its boundaries typically blurred with the notion of *environment*—as clearly emerging from the field of *multi-agent systems* [Weyns et al., 2007]

Situatedness III

Context-awareness: space & time

- mobile computing scenarios have made clear that *situatedness* of computational systems nowadays requires at least *awareness of the spatio-temporal fabric*
- that is, any non-trivial system needs to know *where* it is working, and *when*, in order to effectively perform its function
- in its most general acceptation, then, any (working) environment for a computational systems is first of all made of space and time

! what we call *temporal* and *spatial context*

Situatedness IV

Context-awareness: environment

- more generally, situatedness of computational systems nowadays requires *awareness of the environment* where the systems are deployed and are expected to work
- that is, any non-trivial system needs to understand somehow its working environment, its *nature*, its *structure*, the *resources* it makes available for use, the possible *issues* that may harm the systems in any way

Situatedness V

Knowledge-Intensive Environments (KIE)

- special and nowadays particularly relevant sorts of working environments are the so-called knowledge-intensive environments
- there, *knowledge* available in large amounts, and *distributed* in the working environment, is important (if not essential) for the activities of the systems
- accessing, understanding, and possibly injecting knowledge while interacting (locally) within the working environment is essential for most computational systems to properly perform their activities and achieve their goals

Situatedness & Distributed Systems I

Why distributed systems for situatedness?

- physical distribution of computational systems is essential to cope with the *distributed nature* of many *working environments*
- ... as well as with the need for *situated computations*
- essentially, when the systems requirements mandate for situated computations within a distributed physical environment, situated distributed systems are the only way out
- e.g., disaster recovery scenarios, environmental monitoring, crowd steering, live events...

Situatedness & Distributed Systems II

Openness & scalability

- openness of distributed situated systems is essential to deal with the *unpredictability* of complex environments
- scalability allows distributed situated systems to cope with working environments of *growing complexity*

Pitfalls of Distributed Systems [Tanenbaum and van Steen, 2007]

False assumptions made by first time developer (by L. Peter Deutsch)

- the network is reliable
- the network is secure
- the network is homogeneous
- the topology does not change
- latency is zero
- bandwidth is infinite
- transport cost is zero
- there is one administrator

Such (false) assumptions typically produces all mistakes in the engineering of distributed systems

Pitfalls of Distributed Systems: Remarks

Such (false) assumptions...

... relate to properties that unique to distributed systems:

- reliability of the network
- security of the network
- heterogeneity of the network
- topology of the network
- latency
- bandwidth
- transport costs
- administrative domains

When engineering non-distributed systems, such problems are likely not to show up

Summing Up

- there are good reasons to build up distributed systems
  - several issues, and several opportunities, too
  - systems are inherently *complex*, nowadays, and distributed systems may help hiding some complexity and improving understanding and ease of use
- distributed systems introduces / reveals new dimensions of computational systems
  - when they are ignored, suddenly lead to severe issue
  - to account for them, a new discipline for system engineering is required

References I

Baldauf, M., Dustdar, S., and Rosenberg, F. (2007).
A survey on context-aware systems.
International Journal of Ad Hoc and Ubiquitous Computing, 2(4):263–277.

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2012).
Distributed Systems. Concepts and Design.
Pearson, 5th edition.

Kshemkalyani, A. D. and Singhal, M. (2011).
Distributed Computing: Principles, Algorithms, and Systems.
Cambridge University Press.

Neuman, B. C. (1994).
Scale in distributed systems.
In Casavant, T. L. and Singhal, M., editors, *Readings in Distributed Computing Systems*, pages 463–489. IEEE CS Press, Los Alamitos, CA, USA.

Omicini, A. (2002).
Towards a notion of agent coordination context.
In Marinescu, D. C. and Lee, C., editors, *Process Coordination and Ubiquitous Computing*, chapter 12, pages 187–200. CRC Press, Boca Raton, FL, USA.

References II

Raynal, M. (2013).
Distributed Algorithms for Message-Passing Systems.
Springer Berlin Heidelberg, Berlin, Heidelberg.

Suchman, L. A. (1987).
Plans and Situated Actions: The Problem of Human-Machine Communication.
Cambridge University Press, New York, NYU, USA.

Tanenbaum, A. S. and van Steen, M. (2007).
Distributed Systems. Principles and Paradigms.
Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition.

Weyns, D., Omicini, A., and Odell, J. (2007).
Environment as a first-class abstraction in multi-agent systems.
Autonomous Agents and Multi-Agent Systems, 14(1):5–30.
Special Issue on Environments for Multi-agent Systems.

References III

Zambonelli, F., Castelli, G., Ferrari, L., Mamei, M., Rosi, A., Di Marzo Serugendo, G., Risoldi, M., Tchao, A.-E., Dobson, S., Stevenson, G., Ye, Y., Nardini, E., Omicini, A., Montagna, S., Viroli, M., Ferscha, A., Maschek, S., and Wally, B. (2011).
Self-aware pervasive service ecosystems.
Procedia Computer Science, 7:197–199.
Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11).

Goals & Issues of Distributed Systems

Distributed Systems
Sistemi Distribuiti

Andrea Omicini
andrea.omicini@unibo.it

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

Academic Year 2016/2017