Francesco Russo

JXTAnthill

July 2002

Master's Thesis
Department of Computer Science
University of Bologna

Index

1 The JXTAnthill Project

1.1 Introduction

The Anthill Project aims to define a new framework for *agent−based peer−to−peer* (P2P) distributed systems development, simulation and deployment. As far as P2P systems are concerned, today they are becoming extremely popular since they potentially let us harness the computational power of a huge number of desk−top machines otherwise simply used as dumb client nodes in the actual Internet. In fact the P2P distributed paradigm foresees networks composed by nodes capable of acting both as client and server. What this entails is an improved fault−tolerance and a greater scalability, properties desirable for every distributed system. The Anthill Project is rooted in *complex adaptive systems* (CAS). Complex adaptive systems are dynamical systems, so systems having one or more moving components, characterized by being made up of a huge number of elementary constituents capable of interacting with one another within their environment. Such systems are said to be greater than the sum of their own parts (*holism)* since, despite the simplicity shown by their basic components, they are able to show astonishing behaviors: this is what is called the *emergent behavior* of the system. Complex systems can expose deeply different dynamics according to the degree of interaction that takes place among the system's constituents. Basing ourselves on the former statement, four classes of systems can be found: systems with a *fixed−point dynamic*, *periodic systems*, systems following a *random−like pattern* (*chaotic)*, and systems exposing a kind of behavior which stands between *periodic systems* and *chaotic systems* [TCBON]. These are the systems we are interested in. The Anthill Project is rooted in complex systems since we think there are similarities between peer to peer systems and complex systems: both of them are composed by a number of parts capable of interacting with one another. Even further CASs offer several appealing properties such the total lack of centralization and control, an amazing ability to tolerate rapid changes in their own structure, and a kind of self−organization towards a global configuration. One fundamental and surprising thing is that in the critical area between periodic and chaotic systems, reside systems known to be capable of universal computation. What this means is that relying on this new computational paradigm it is now possible to solve problems in an inherently decentralized way. Now it is evident why borrowing concepts and ideas from CAS, for developing a framework for P2P applications development. The Anthill Project is based on the usage of *mobile agents* as well. In the Anthill terminology, mobile agents are called *ants*. Ants societies are a real example of CAS we can find in nature. This CAS instance is of great interest since it has always lent itself to be easily implemented for successfully solving many optimization problems. So, in the Anthill framework, virtual ants are in charge of travelling across the network according to a simple set of rules for achieving their goals. In the Anthill Project, emergent behavior manifests itself as swarm intelligence whereby the collection of simple ants of limited individual capabilities achieves “intelligent” collective behavior [BMM−09−01]. Along with this thesis we propose an Anthill Project implementation − an Anthill binding − based on the JXTA Technology: JXTAnthill. JXTA is an open source project promoted by Sun Microsystems, Inc. What Project JXTA defines is a platform for developing P2P applications guaranteeing the following properties: *interoperability*, *ubiquity*, *platform independence* (see Chapter 4 for details). JXTAnthill is the result of the integration of the Anthill Project with the JXTA Technology. Basing our Anthill binding on JXTA we have avoided to deal with *peer group establishment* and *management*, *peer monitoring* and *firewall and NAT traversing* issues [PJJFPG]. Furthermore JXTA provides basic security mechanisms such as scure communication channel implementation based on Transport Layer Security (TLS) Version 1. The TLS specification is currently under development by the

Internet Engineering Task Force (IETF) Network Working Group [SPJ]. For all these reasons we thought of JXTA as being the most suitable technology for developing our first Anthill binding. For demonstration purposes we propose an example application developed on top of JXTAnthill. This is a P2P document sharing application which uses a virtual ant species named *Gnutant* for implementing the document sharing service. This ant species mimics the behavior of two of the most noticeable protocols devised for P2P content sharing, *Gnutella* and *Freenet*, mixing the pros of both of them.

This thesis is organized as follows:

- *Chapter 2.* The first chapter deals with complex systems. *Cellular Automata* and the *Game of Life* are introduced for showing the different dynamics such systems expose, and for giving an informal demonstration of how the *Game of Life* is capable of universal computation. Insect societies are taken into account as natural instances of complex systems, and interesting examples about virtual ants and virtual termites are proposed.
- *Chapter 3.* This chapter introduces P2P distributed systems. It is shown how distributed systems have evolved in the years from a purely centralized approach to a completely decentralized one. Three cases of study are proposed: *Gnutella*, *Freenet*, *SETI@home*.
- *Chapter 4.* This chapter covers the JXTA Technology topic carrying out a detailed survey over its architecture, the set of protocols it defines and their behavior, the message format etc.
- *Chapter 5.* This chapter defines the aims of the Anthill Project and introduces concepts fundamental to the reminder of the thesis.
- *Chapter 6.* This chapter presents a detailed architectural description of both the Anthill Project and the JXTAnthill binding.
- *Chapter 7.* This chapter is concerned with implementation details of the JXTAnthill binding

- *Chapter 8.* This chapter introduces a sample application based on a virtual ant species called *Gnutant*. This ant species implements a file sharing service.

2 Complex Systems

2.1 Introduction

One of the inspiring concepts of the Anthill Project has surely been what nowadays we know as *Complex Systems*. This class of systems exposes many interesting properties that make them appealing for anyone looking for inherently decentralized solutions to a wide variety of problems. Before starting out with Complex Systems and their properties, it could be useful to carry out a survey about *Dynamical Systems* since they are closely related to Complex Systems*.*

A *dynamical system* might be informally defined as something having a dynamic component, or equally anything which has motion. Even if this definition could sound simplistic to someone, it is really what is needed in order to identify such a system [TCBON].

When analyzing a dynamical system, we are interested in describing the moving components of the system as well as the set of rules governing its dynamics. Based on these considerations it is possible to draw a grading of dynamical systems exposing similar properties. This classification envisages four different types of dynamical systems, which are brought together by the same kind of dynamic. The simplest one has a so called *fixed point dynamic*. This means that the system will eventually enter in a state from which it will never get out. For example one could think of a ball rolling on a plain surface until it has no more momentum to carry on going. The second class is composed by systems having a *periodic dynamic.* For example such a system could be an idealized pendulum. The next class is about systems characterized by the so called *quasiperiodic dynamic*. We have a quasiperiodic motion when the system roughly behaves always in the same way, where “roughly” means that the system will experience states always close to already experienced ones, but will never repeat anyone of them. The types of dynamical systems described so far all expose a certain degree of predictability: being aware of the system’s initial configuration and its initial state, it is possible to predict its future configurations. But there is also a fourth class envisaging systems that do not expose predictability: the *Chaotic Systems*. These systems differ from anything else since there are no specific patterns in their behavior, and due to their complexity any long term prediction is impossible to achieve. The amazing thing to highlight is that such systems are not the exception but the very huge majority in nature, and what previously was thought to be a *random phenomena* is nowadays known as chaotic. So, the main intuition here is that in such systems it is false that there are no rules and that the overall behavior is completely random, instead the set of involved rules and variables is just too complex and prediction is admitted only in the short term. Now, what stands behind this digression? Complex systems expose all the properties listed so far, the full spectrum of possible behaviors, and so can be thought as a particular instance of *Dynamical Systems*.

2.2 Complex Systems

A complex system is a dynamic system itself, but characterized by a fundamental property:

Complex Systems are things that consist of many similar and simple parts. Often the underlying behavior of any of the parts is easily understood, while the behavior of the system as a whole defies simple explanation [TCBON].

Complex systems are characterized by the presence of a set of elementary parts capable of interacting with each other. They can be said to be elementary since they are not supposed to expose any kind of intelligence or cleverness.Generally they behave in a very simple way and so such a system could even be described by a relatively reduced set of rules. The most interesting thing about complex systems is tightly related to their building blocks' simplicity: unless their single simple parts' inability to perform amazing tasks if taken as a unit, the overall system's behavior can sometimes astonish the observer. That's what is called the *Emergent Property* of the system. By changing the type and form of interactions that take place among the parts of such a system, it becomes globally goal−seeking while only local information is passed around by the parts. In other words we might say that the whole system seems to be greater than the sum of its parts (*holism*). This is really encouraging since it means that a collective form of computation can take place without an explicit global algorithm.

2.3 Cellular Automata and Universal Computation

In order to deal with complex systems many different formalisms have been devised. In this chapter we introduce and use *Cellular Automata* for describing the relationships existing between complex systems and the four different dynamics formerly discussed. Cellular automaton have been devised by John von Neumann in the 1940s, for studying the reproduction process. The simplest CA we can refer to is the *one−dimensional cellular automa:*

Imagine a linear grid that extends to the left and right. The grid consists of cells that may be in only one of a finite number of states, *k.* At each time step the next state of a cell is computed as a function of its neighbours local in space [TCBON].

So if we denote the radius of a cell's neighbourhood with *r*, it will embrace a set consisting of *2r + 1* cells. The function denoting the state of a cell *i* at time *t* will be:

$$c_i(t)$$

Taking into account the values: *k = 2*, *r = 1*, one possible rule table for a one dimensional CA would be:

$c_{i-1}(t)$	$c_i(t)$	$c_{i+1}(t)$	$c_i(t+1)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Table 2.1 Sample rule table

This table's content could be compressed in the following rule: "If all the cells in a neighbourhood are on or off, then the next state is off; otherwise, the next state will be on". In 1980s Stephen Wolfram carried out a CAs classification and the result showed a lot of similarities with the dynamic systems grading formerly exposed. Infact still four classes are envisaged:

- **Class 1**. CAs in the first class always evolve to a homogeneous arrangement, with every cell being in the same state, never to change again.
- **Class 2**. CAs in the second class form periodic structures that endlessly cilce through a fixed number of states.
- **Class 3**. CAs in the third class form random−like patterns that are a lot like the

static white noise in a bad television channel.

- **Class 4**. CAs in the fourth class form complex patterns with localized structure that move through space in time. The patterns must eventually become homogeneous, like Class 1, or periodic like Class 2.

Reasoning by analogy it is easy to find a kind of mapping between Wolfram's classification and dynamical systems classification, at least for classes ranging from first to third. They can be respectively mapped to a fixed point dynamic, a periodic dynamic and a chaotic dynamic. What defies our intuition are Class 4 CAs since they seem to expose all of the behaviors we can find in the other three classes. They seem to be somewhere between class two and class three, so tracing an ideal CA rule space we should proceed in the order of Class 1, Class 2, Class 4 and Class 3. It is even possible to view this classification on a more formal basis using what is known as the *Langton's Lambda Parameter*.
This parameter helps us associating a generic CA to a scalar value, obtainable being aware of what the CA's rule set is, and gives us information about the system's behavor. Having λ close to 1 entails that the system will expose a chaotic behavior; values closer to 0 charachterize systems with a fixed−point like dynamic. Given a CA with parameters *k* and *r* (respectively the number of possible states and radius of a cell's neighbourhood), the total number of entries in the rule table will be equal to k^{2r+1} .
Among these rules there is a potentially empty subset envisaging those ones leading to what Langton identifies as a *quiescent state*. Such a state is a state that we can define as being inactive or off. Letting the number of rules mapping to a quiescent state be n_q we define the Langton's Lambda Parameter as follows:

$$\lambda = \frac{(k^{2r+1} - n_q)}{k^{2r+1}}$$

or equally, letting N be the total number of rules:

$$\lambda = \frac{(N - n_q)}{N}$$

λ's value ranges within the interval [0..1]. If λ is equal to 0, the CA has a fixed point like dynamic since all rules map to the quiescent state. Otherwise if it is equal to 1, no rule would map to any quiescent state, and so the CA's behavior should sound chaotic. With λ equal to $(1 - 1/k)$ all states will be equally represented in the rule table.

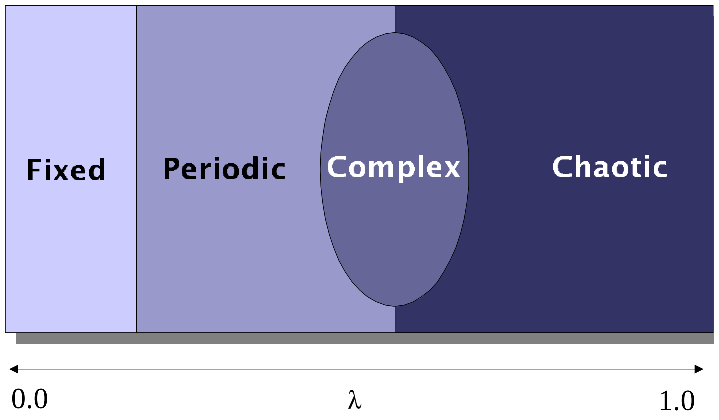


Figure 2.1 Representation of CA rule space characterized by the λ parameter.

Between λ = 0 and λ = $(1 - 1/k)$ has been experimentally proved that lie all the cellular automata belonging to Class 2 and Class 4, with Class 4 CAs having a greater λ then Class 2 CAs in average which confirms Wolfram's intuition. The area in which Class 4 CAs reside is known as Langton's Critical Area, and it is known that some CAs belonging to this class are capable of universal computation. One of these CAs is known as "The Game of Life". This is a two dimensional cellular automaton devised by John Conway in the late 1960s. It is described by the following set of rules [TCBON]:

- If a living cell has less than two neighbours, then it dies.
- If a living cell has more than three neighbours, then it dies.
- If an empty cell has three living neighbours, then it comes to life.
- Otherwise (exactly two living neighbours), a cell stays as is.

Now, what do we need in order to state the “Game of Life” is capable of universal computation? We should show that it allows us to build at least two logical primitives, i.e. NOT and AND. Before showing how this is possible it is necessary to introduce three simple classes of object we can find in Life. Firstly there are simple static objects, configurations that do not change their shape and position over time. In figure 2.2 there are two patterns of this kind. Secondly there are periodic structures as well, which we may need both for counting and for synchronization purposes (Fig. 2.3). Last but not least there is a class of moving objects. They are generally classified as *gliders* if they move one diagonal space in four time steps, or *fishes* when moving two horizontal or vertical squares in four time steps. Moving objects like these and making them collide with each other in a precise manner we obtain a new class of objects, the *Glider Gun* which is an object emitting gliders at regular intervals of time.

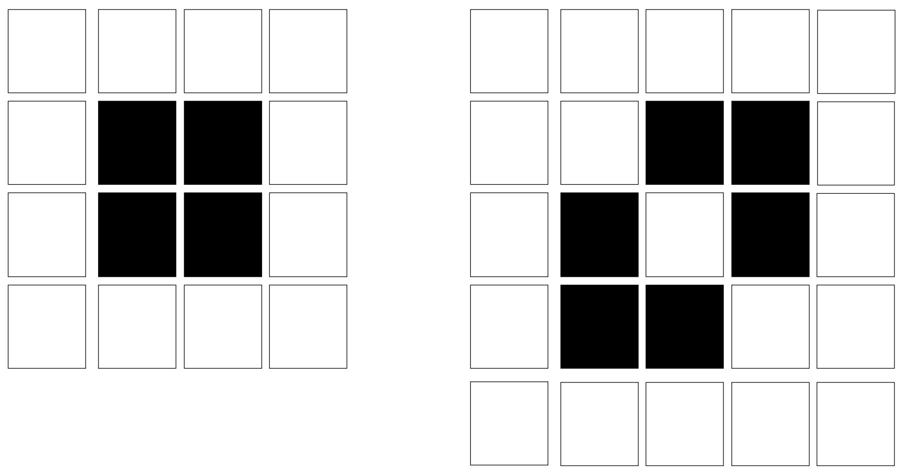


Figure 2.2 Static objects in Conway’s Game of Life

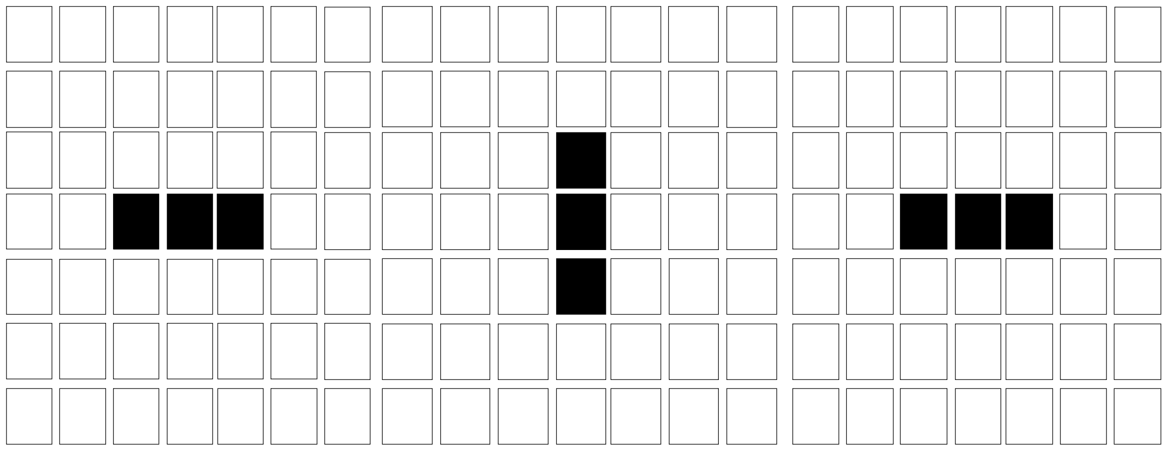


Figure 2.3 Periodic object in Conway’s Game of Life

Let us now see how it is possible to obtain the NOT operator starting from these building blocks. In figure 2.4 we can see a glider gun named $\mathcal{G}$ and an information source named $\mathcal{S}$. $\mathcal{S}$ will emit a glider only when there is a 1 in the input data stream while the glider gun $\mathcal{G}$ will be always emitting a continuos stream of gliders. When in the source stream there is a 0, the glider coming from $\mathcal{G}$ will be able to reach the sink, while when the source stream contains a 1, $\mathcal{S}$ and $\mathcal{G}$ will both emit a glider and these two gliders will collide annihilating each other. The surviving gliders coming from $\mathcal{G}$ will exactly correspond to the empty locations in the glider emitted by $\mathcal{S}$ (remember that $\mathcal{G}$ emits a continuos stream of gliders) so the logical negation of the source. The AND circuit is based on the same logic and it is represented in figure 2.4: what we want is to have a glider arriving at $\mathcal{S} \wedge \mathcal{S}'$ only when both $\mathcal{S}$ and $\mathcal{S}'$ are 1. Since we have an AND and a NOT circuit we can state that theoretically the “Game of Life” is capable of universal computation, which is equivalent to say that we have complex systems capable of universal computation.

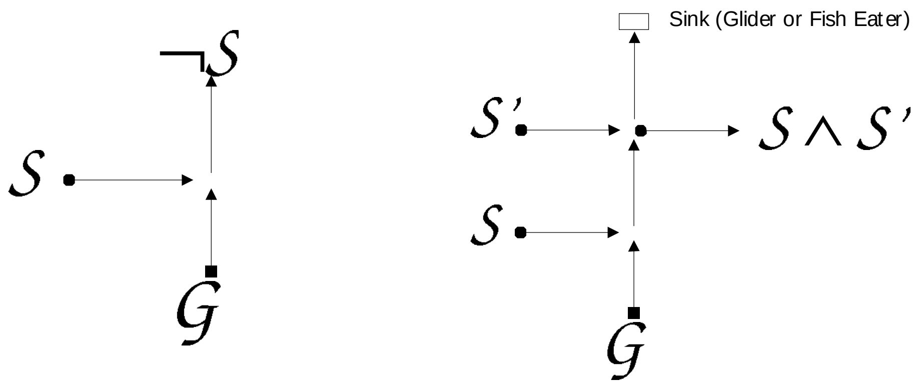


Figure 2.4 Logical NOT primitive and logical AND primitive in Life

2.4 Complex Systems as Insect Colonies

As stated before, complex systems are very common in nature, and observing natural systems can often lead to amazing intuitions. In this respect insect colonies have been widely studied in the last years, since they seem to show many similarities with complex systems. Looking for the possible points of contact between complex systems and insect colonies, we immediately find out that in both cases there are many simple components whose behavior can be characterized as being deeply specialized in some task execution. Furthermore there is absence of a global knowledge about both the whole system and its parts, an high interaction between the system's elementary components, and an high grade of parallelism in the execution, since agents can act independently of each other. The most interesting fact is that insect colonies are able to produce remarkable results that go far beyond the scope of any individual insect, and that is exactly what is known as *emergent behavior*. Now, we need to add a new fundamental concept: the notion of *Autonomous Agent*.

An *autonomous agent* is a unit that interacts with its environment (probably consisting of other agents as well) but acts independently from all the other agents in that it does not take commands from some seen or unseen leader, nor does an agent have some idea of a global plan that it should be following.

Agents like this, interacting as suggested in the definition, are able to create globally−order systems: for this reason they are called *self−organizing*.

2.5 Self−Organization, Insects and Virtual Ants

Self−Organization is defined as a set of dynamical mechanism whereby structures appear at the global level of a system from interactions among its lower level components. The rules specifying the interactions among the system’s constituent units are executed on the basis of purely local information, without reference to the global pattern, which is as emergent property of the system rather than a property imposed upon the system. A self−organizing behavior is characterized by a well known set of properties [SOSI]:

- **Positive Feedback**: that is the ability of the system to favour one solution rather than another, trying to adopt more extensively behaviors that have proved to be more successful.
- **Negative Feedback**: counterbalances positive feedback and helps to stabilize the collective pattern.
- **Fluctuation**: only positive and negative feedback would not be enough for a system to find new solutions and improve its overall behavior. There must be some sort of randomness since it enables the discovery of new solutions unknown in the past history of the system.

The following examples are relative to natural complex systems, used as a starting point for showing how they can be artificially reproduced taking advantage from the properties they expose for solving different kind of problems.

2.5.1 Foraging in Ants

This is one of the most known experiments carried out in this field. The ant species involved is the *Linepithema humile* [SOSI]. In this experiment the nest had been separated from the food source by a bridge with two equally long branches. It had been observed that letting the ants go from their nest to the food source and back, once the branch was reached an ant randomly chose where to go. Now it is necessary to point out that ants, like any other insect species, are known to lay pheromone trails down while travelling to a food source. That is probably done in order to be able to come back to the nest and for instructing the other ants in how to reach the brand new discovered food source. The interesting fact is that using two different branches the ants were able to find out which was the shorter one, and kept on choosing it more likely than the other. This global emergent behavior is due to the fact that initially ants select at random which branch to traverse back and forth. This implies that in the same amount of time more pheromone will be laid all along the shortest branch, and this will draw ants on this same path even in the future. Laying down on the traversed path the pheromone trail is a form of positive feedback. The negative feedback is given by the evaporation of the pheromone itself, and we still have fluctuations since ants may act at random.

2.5.2 Construction of pillars in Termite Colonies

This example involves the *Macrotermes* termite species [SOSI]. While building their pillars up, it has been observed that they follow a two−steps process:

- *Non−Coordinated Phase*: each termite lay randomly down soil pellets impregnated with pheromone.
- *Coordination Phase*: this is the phase during which the pillar emerges.

What really matters is that the coordination phase does not start unless both the deposit reaches a critical size and the group of builders is sufficiently large. Even in this example, the positive feedback is given by the pheromone−soaked soil pellets: in this way termites are encouraged to deposit more pellets where they feel there is more pheromone, making more likely the coordination phase. But if the number of termites in the building team is simply not enough the pheromone is bound to rapidly evaporate, originating a negative kind of feedback and making impossible the coordination phase start up. Fluctuations are given by changes in the termites population size: this affects the probability to have success in the future.

2.5.3 Dead ants corpses clustering in Ant Colonies

The clustering phenomena has been studied in the *Pheidole pallidula* ant species [POKA]. The scenario is a two dimensional space over which are spread corpses of dead ants. The space is the ant colony’s foraging area. It has been observed that an ant behaves roughly as follows: it takes a random walk through the two dimensional area and when it runs into a dead body or a dead item, it picks it up with a given probability $\mathcal{P}$. $\mathcal{P}$ tends to increase if near the discovered body there are no other dead items, so if the body does not belong to an already existing cluster; $\mathcal{P}$ tends to decrease otherwise. So the global behavior of the colony results as aimed to the clustering of objects. Even in this case pheromone plays an important role since trails leading to already existing clusters can improve the overall process.

2.5.4 Virtual Termites

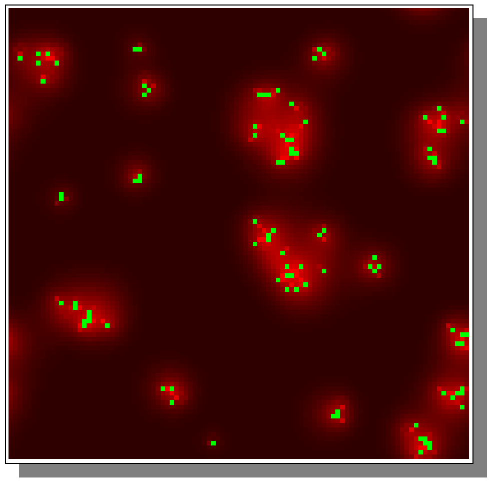
After having observed what we can find in nature, our interest gets focused on what we are really interested in: can we reproduce such a behavior having self−organization and emergent properties for solving problems in a truly decentralized manner? As our first example we refer to a virtual termites species whose aim is the clustering of resources in a two dimensional space. The task is the same performed by he *Pheidole pallidula* ant species. The behavior of this theoretical virtual termite devised by Mitchel Resnick is based on three simple rules [TCBON]:

- Wander around aimlessly, via a random walk, until bump into a wood chip.
- If the termite is carrying a wood chip, it drops the chips and continues to wander.
- If the termite is not carrying a wood chip, it picks up the one that it bumped into and continues to wander.

The proposed rule set is very simple; nevertheless, the resulting behavior is amazingly complex. This system has been formalized using a cellular automata which has been iterated hundreds of thousands of time steps. The result has been the birth of a kind of order in the primordial chaotic space where resources are gathered in clusters equally distributed over the area. This termite species could be obviously modified in order to obtain different results or in order to improve the overall performance.

2.5.5 Swarm and Heat−Bugs

Swarm is a software package for multi−agent simulation of complex systems, originally developed at the Santa Fe Institute. The basic architecture of Swarm is the simulation of collections of concurrently interacting agents. The world is represented as a two dimensional grid over which autonomous agents, the heat−bugs, can move. Each cell in the grid has a property, its heat, which changes over time evaporating and diffusing. In figure 2.5 this property is represented using different red color gradations. Heat−bugs are the green dots, and they simply aim to find a place in their two−dimensional environment where there is a temperature



fitting their needs. How does this system evolve? Each time step every heat−bug moves one step to a nearby location looking for the place that makes it happiest, and emits a bit of heat. Since any heat−bug by itself can be warm enough, in the long term the population will tend to cluster together. In order to find some mapping between heat−bugs and the real world we are living in, this problem could be thought as an optimization problem, where one wishes to maximize each heat−bug happiness. As shown in figure 2.6 we can achieve this goal even dealing with a system based on simple rules. Even in this case the global state of happiness seems to be an emergent property.

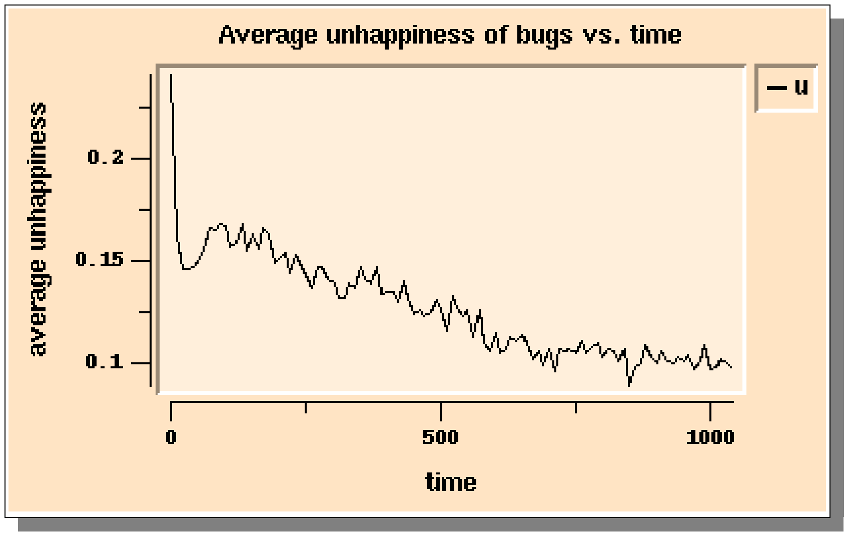


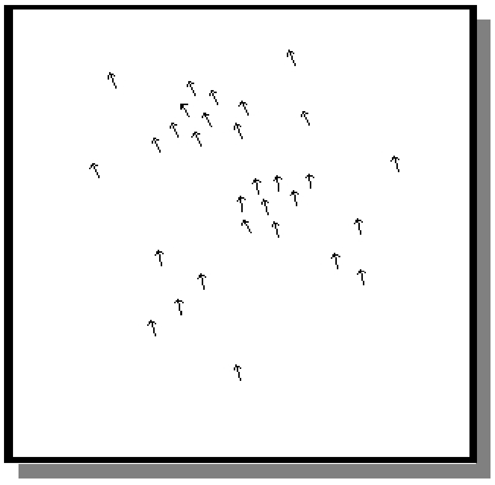
Figure 2.6 Heat−Bugs average unhappiness over time

2.5.6 Boids: Flocks of Virtual Birds

The last example is about what is usually called a *boid*. A boid is a virtual bird whose behavior is dictated by four simple rules. Boids have been devised in order to study the dynamics ruling the flocks of birds. Many bird species are often capable of such astonishing evolutions that one might wonder how they can do that. Apparently this global behavior can be reproduced creating boids who simply follow the given list of rules:

- **Avoidance**: Try and reduce the chance of collision with other boids in the flock, dynamically changing the distance from them.
- **Copy**: Fly following the general direction the flock is moving by averaging the other boids' velocities and directions.
- **Center**: Try to be in the most inner area of the flock, for safety reasons.
- **View**: Move laterally away from any boid that blocks the view.

Having a number of boids flying over the two dimensional space it can be observed a global behavior pretty close to the one showed by flocks of birds in nature: the boids reach the typical "V" formation. So we can state this is again an emergent property of the system we defined only tracing one boid's behavior, which may even seem selfish since every agent tries an maximize its own degree of satisfaction.



2.6 Conclusions

What we argue at this point is that with complex systems and autonomous agents, each individual behaves competing and cooperating with the others according to the set of rules defining what it is bound to do. Despite how these rules might seem, selfish oriented or based on pure cooperation, the interesting thing is the overall behavior we get back. In the boids example, the set of rules yield an emergent property which looks a lot like bird intelligence. We can consequently identify many different layers defining a system. Each layer is characterized by properties and behaviors potentially different from the ones we might observe at the lowest levels of the system itself. The more our attention gets focused on the deepest layers, the simpler components can be found: agents whose behavior can easily be defined by a simple set of rules. Adding recursion, multiplicity and parallelism to these elementary building blocks we can get emergence and self−organization, and use these systems and their properties in order to solve problems in a purely decentralized way.

3 Peer−to−Peer Computing (P2P)

3.1 From Distributed Systems to P2P

As networking is becoming a widespread reality, distributed systems are replacing centralized systems where they begin to show their weaknesses. This is due to the fact that distributed systems expose a set of interesting properties making them more suitable than any other known solution for solving a huge variety of problems. Before going on it is necessary to provide a definition of distributed systems:

A *distributed system* is a collection of sequential processes and a network capable of implementing some sort of communication among the given processes [DS].

This definition does not clarify how processes should communicate with one another, or which kind of communication pattern the underlying network infrastructure should be able to implement and guarantee since we are not interested in such details. Switching from a purely centralized solution to a distributed system involves facing and solving a set of new non−trivial problems. Anyway, choosing a distributed system we gain especially in terms of *reliability*, *performance* and *scalability.* Having a centralized system means having a single node in charge of doing the whole job, and if this node crashes the service will not be available anymore. Since we cannot be completely sure a machine will never fail, what we can do is try and reduce as much as possible the overall probability of failure, and one way for achieving this goal is known as *replication in space*. This implies that having *n* machines cooperating in some way for performing the same task, reduces the probability for the overall system to become unavailable.

That is why distributed systems improve reliability. We can even gain something in performance imposing a specific cooperation logic within the system, for example aiming to equally balance the load among the nodes composing the cluster, or delegating specific tasks only to specific nodes. One of the main disadvantages of a centralized system is that it cannot easily grow. In this case scalability is strictly limited by the capacity of the server, while in a distributed system scalability is improved by the ability of dynamically adding new nodes to the system. Anyway distributed systems are not a panacea by themselves. There are many different kinds of topologies characterizing distributed systems, each one exposing pros and cons that make a choice more or less adequate to designers' or developers' needs.

3.2 Distributed Systems Topologies

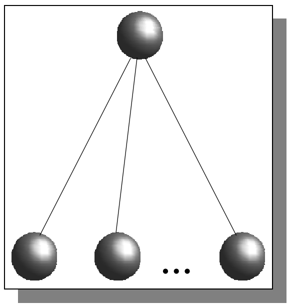
There are four different distributed system topologies, namely *centralized*, *ring, hierarchical* and *decentralized*. Each of them exposes merits and weaknesses at the same time, and that is the main reason why hybrid topologies are often better. Appropriately mixing them enables to compensate for lacks in one topology profiting from the valuable properties we can find in another one. Since topology might be considered at different levels (physical, logical) for this analysis topology is thought in terms of information flow [DST]. In order to compare each topology pattern with the others, the following seven properties are evaluated:

- **Manageability**: How easy is to keep the whole system up and running?
- **Information Coherence**: How authoritative is information in the system? If some of data is found in the system, is that data correct?
- **Extensibility**: How easy is adding new resources to the system?

- **Fault Tolerance**: How does the system tolerate crashes?
- **Security**: Security is a truly wide topic and goes from avoiding intrusions to preventing people from injecting undesired information into the system etc.
- **Resistance to lawsuits and politics**: How hard is for an authority to shut the whole system down?
- **Scalability**: How large can the system grow?

3.2.1 Centralized

Centralized systems are surely the most common pattern we can find in networked systems nowadays. That is the so called client/server model, and has been used for deploying databases, web servers, and many other simple distributed systems for years. The whole logic of the system resides within one single node, the server, whose aim is performing tasks on behalf of client machines connected to it.



- *Manageable*: Yes. Since centralized systems consist of a single node, any maintenance order can be easily performed without major complications.
- *Coherent*: Yes. As there is a single point where the logic of both the information and the system reside, it is relatively easy to keep the content of the system coherent.
- *Extensible*: No. If having a single node can improve manageability and coherency, this characteristic disadvantages extensibility: resources can only be added to the single node composing the system, and the node obviously exposes its own capacity extents.
- *Fault−Tolerant*: No. As stated before, systems consisting of one single node are less reliable than systems made up by a cluster of cooperating machines.
- *Secure*: Yes. It is surely easier to protect one single node than a system

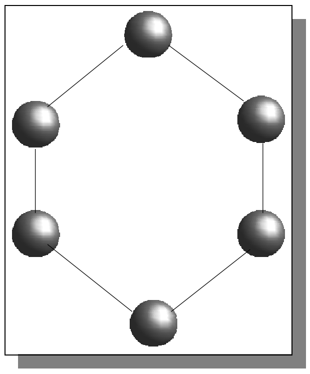
exposing a complex structure, so systems like this can be thought as being relatively secure.

- *Lawsuits−Proof*: No. In order to shut the whole system down, forcing the disconnection or the crash of the centralized node is sufficient.
- *Scalable*: Not completely. Centralized systems are not expected to scale over a certain threshold due to their physical extents: maximum number of concurrent processes admitted, maximum number of simultaneous connections, storage capacity etc.

Due to fault tolerance related limits, generally such a system is improved by using a cluster of machines transparently acting as a single node (see the next topic) or by using back−up nodes ready to replace the crashed one if necessary.

3.2.2 Ring

As stated before, sometimes it is a good idea replacing a centralized server with a cluster of machines, mainly in order to improve reliability. Generally these clusters are arranged in a ring topology, with all the nodes acting transparently as a centralized server.



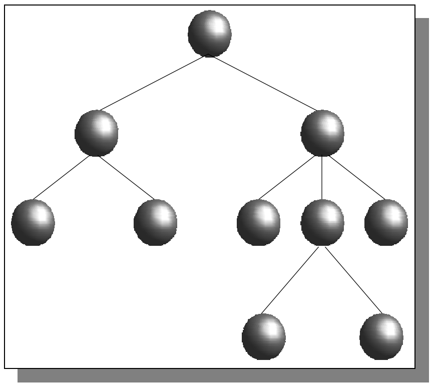
- *Manageable*: Yes. Such a system often has a single owner, and is managed like a centralized system would be. So, as far as manageability, coherency, and security are concerned, these systems do not differ from a centralized topology too much.
- *Coherent*: Yes. Since the whole system belongs to one owner or a single organization, its coherency should be easily granted.
- *Extensible*: No. The single−owner restriction means a user will need the owner's endorsement for adding a resource into the ring.

- *Fault−Tolerant*: Yes. This is the main advantage of a ring topology over a centralized one. The replication in space improves the system’s reliability.
- *Secure*: Yes. Protecting a cluster of machine instead than one single node, does not require a deeply greater effort.
- *Lawsuits−Proof*: No. If someone would shut the system down, he had just to disconnect the cluster from the network, which is pretty easy to do.
- *Scalable*: Yes. Ring systems scale better than centralized ones: we can improve system’s performance just adding a new node to the cluster.

3.2.3 Hierarchical

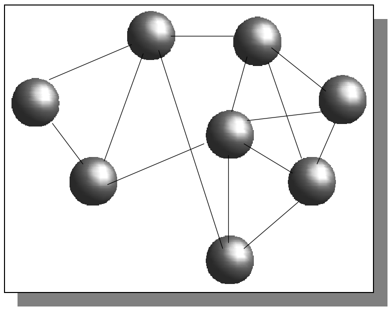
This pattern is the most common in the Internet today. By the name, it is easy to understand what this topology looks like. One of the most known hierarchical systems is maybe the Domain Name Service (DNS), where authority flows from the root name−servers, to sub−level servers. Another interesting example of hierarchical system is the Network Time Protocol (NTP). NTP has over 175000 hosts with most of them being two or three links away from a root time source.

- *Manageable*: Partially. Having a well defined architecture might bring someone to guess hierarchical systems are highly manageable. That is not completely true: the wider the system becomes, the harder it gets to manage it easily.
- *Coherent:* Partially. Coherency is usually achieved with a cache consistency strategy.



- *Extensible*: Yes. Any host can easily add data to the system, but it is important to take into account how data propagates through the system.
- *Fault−Tolerant*: Partially. Their are not completely fault tolerant, since the system's root still remains a single point of failure.
- *Secure*: Partially. The growing complexity of these systems renders securing them harder than securing a centralized system or a system characterized by a pure ring topology.
- *Lawsuits−Proof*: Partially. Having a single point of failure (the root) renders these systems rather fragile.
- *Scalable*: Yes. Hierarchical systems scale reasonable well. As an example consider the DNS which has scaled from a few thousand hosts to hundred of millions over the last 15 years.

3.2.4 Decentralized



In such a topology there is not a well defined architectural pattern the overall system should reflect. All the node are only required to be able to interact symmetrically with one another, playing equivalent roles. Maybe one of the most important examples of decentralized systems is the Border Gateway Protocol. BGP has been devised in order to replace the Exterior Gateway Protocol (EGP) when it started out showing its weaknesses: EGP forced a treelike topology onto the Internet and it did not allow for the topology to become more general [CNET]. So, this shows us how hierarchical structures can sometimes become no more useful due to their inherent lack of elasticity, and in these cases decentralized solutions can surely fit better.

- *Manageable*: Partially. It depends on how huge the system is and whether the system is managed by a single authority or not. This means that if the decentralized system is made up of nodes belonging to individuals who get together for some common purpose or interest, then each node will be independently managed by the user it belongs to. On the contrary if a single organization manages a widely distributed system, it would not be easy anymore to accomplish this task.
- *Coherent*: No. Since there is not a centralized management, coherency is mainly up to users, and there is no control over their activity.
- *Extensible*: Yes. Extensibility is one of the major properties exposed by decentralized systems. Each node can freely add resources to the system, but the propagation scope of each update strongly depends on the underlying protocol.
- *Fault−Tolerant*: Yes. Decentralized system are extremely fault tolerant: each node acts independently both as client and server, so a failure does not deeply affect the overall behavior, especially when resources are replicated in space through the system.
- *Secure*: Depends. It depends on how the system has been designed. If we adopt a restriction policy for users joining the system, along with secure communication channels, the system might reach a good level of security.
- *Lawsuits−Proof*: Strongly. Not having any form of centralization, it is impossible to shut a purely decentralized system down.
- *Scalable*: Partially. As we are going to see this property depends on the underlying protocol.

3.3 P2P

Sometimes choosing the right topology is not enough. What makes the difference is the way in which nodes interact with one another. A new paradigm for building distributed applications called *peer to peer* has recently gained particular attention, not only among researchers. Let us now give a definition of what P2P is:

Peer−to−Peer (P2P) computing is the sharing of computer resources and services by direct exchange between systems. These resources and services include the exchange of information, processing cycles, cache storage, and disk storage for files.

P2P computing is not as new as one might think. About thirty years ago institutions were working on architectures that now would be labeled as *peer−to− peer*. All the nodes were able to perform routing, to act both as server and client for services like ftp or telnet and so on. From that days on, Internet was shaped in a more hierarchical way, and that is the Internet we know today. But all along the way from 1960s to our days many things have deeply changed. Today we have a huge number of nodes connected to the network, most of them intermittently, and all having a great amount of unused resources (CPU cycles, storage, content management etc.). The current client/server model does not let us benefit from what is usually called "*the dark matter*" of the Internet, the amazingly huge set of resources available out there [P2PAM]. P2P computing mainly has two really interesting advantages:

- First, it let us obtain remarkable results in a cheap way. P2P let us collect resources scattered all around the globe, and coordinate them for accomplishing one single task. Doing something like this with a centralized approach would be inadmissible (see the SETI@home Project case study at the end of this chapter).

- The second important issue is about bandwidth. By decentralizing resources and redirecting users to other users' desk−tops for downloading resources, Napster, the most famous P2P application, reduced the load on its servers to the point where it could cheaply support tens of millions of users. A company called CenterSpan has roughly estimated that gathering together the whole content of P2P systems like Gnutella and Napster, and having it delivered from one central server, one should need something like 25000 T1 lines costing 25 million dollars a month. In short peer to peer cannot only distribute files, it can also distribute the burden of supporting network connections eliminating bottlenecks at central sites and equally importantly at their ISPs [P2P4A].

3.4 The Five Models of P2P

3.4.1 Atomistic Model

This is the purest form of peer−to−peer computing. There is no mediation by servers, so peers must be able to establish communication channels by themselves. An atomistic approach may be applied for example if each user/peer would always be associated with the same address.

3.4.2 User Centered Model

User centered applications utilize a *directory* (distributed or located on a single server) to provide an efficient way for users to make connections with one another on the network. Once two clients are connected, the system will behave in a purely peer−to−peer manner. The centralized directory acts just as a rendezvous place, a well known point where peers can find each other. Real applications based on this model are ICQ and AOL Instant Messenger.

3.4.3 Data Centered Model

Data centered peer−to−peer applications allow users to search and access data and content held on other users' systems. Usually applications belonging to this class act in the following way: when a peer becomes member of the group, an index containing the set of resources it is willing to share is added to the group's global index, which may be distributed or not. Queries are executed over this global index while any other task is carried out in a completely decentralized manner. More advanced implementations of this model can be proposed, especially in order to overcome the need of a global index (autonomous agents performing searches according to some logic − the *Anthill Project*).

3.4.4 Web Mk 2 Model

Web Mk 2 model [GC] is a convergence of all the above models with current Web architectures and infrastructures: today's browsers will evolve into user−configurable workspace managers, which integrate the three types of P2P interactions previously described into task−specific work environments. P2P applications belonging to this model will heavily rely upon agents (software−bots). Applications and data will reside in multiple locations, with users being able to interrogate information held on servers, other user systems, or even bypass server−based computing when desired transparently.

3.4.5 Compute Centered Model (Distributed Processing)

With distributed processing we have an unusual kind of peer−to−peer computing: a node, or a cluster of nodes, is used for splitting the whole job into small chunks that will be given to nodes in charge of performing the whole processing. The nodes can even be geographically distributed, and they are used on an opportunistic basis. This approach is truly appealing since it let us harness massive parallel CPU cycles from under−utilized and low−cost clients instead than buying expensive parallel machines whose performance might even be worse than the one exposed by systems based on distributed processing. An example is the SETI@home project (see the related case study at the end of this chapter).

3.4.6 Formal and Informal P2P Applications

The last necessary classification of P2P applications concerns security. We can talk about *formal* and *informal* peer−to−peer architectures. It is not accurate assuming that all peer−to−peer applications are inherently insecure and encourage wild access to data and resources. peer−to−peer models can be designed to be:

- **Formal**: Strictly controlled access to resources, with protocols monitored at the network level (e.g. NextPage and Groove).
- **Informal**: Uncontrolled access to resources, so environments not governed by formal corporate policy (e.g. Napster and Gnutella).

3.5 Analysis − P2P is not enough

3.5.1 Case Study 1: Gnutella and Freenet

As stated before, choosing the right architectural model is not enough. What P2P promises to bring us is *high availability*, *fault tolerance* and especially *scalability*. In order to enjoy all these properties, P2P has to be absolutely sustained by something more: a suitable protocol. Let us take into account an interesting example: Gnutella. Gnutella is a protocol for distributed search started by Nullsoft in March 2000 and then closed due to its potential use for copyright infringement. Then, when the protocol was reverse engineered, Gnutella became an open source project and started to spread. A Gnutella network is roughly made up by peer nodes called *servents* (*serv*er, cli*ent*), which can interact with one another through message exchange. There are five message types:

- PING: this message is meant for testing one peer's state, so whether it is alive or not.
- PONG: this is the reply message following a PING message.
- QUERY: this message type is used in order to query the Gnutella network.
- QUERYHIT: if a servent receives a QUERY message and it is able to satisfy the query, a QUERYHIT message will be sent back to the requesting servent. This reply message contains enough information for the requesting node to download the required resource.
- PUSH: A mechanism that allows servents behind firewalls to serve files.

Consider now how a generic search session evolves. When a servent needs a resource, it will send out a QUERY message to all its known neighbours: in other words the set of nodes it is already connected to. Every servent receiving such a message firstly tries to satisfy the incoming query, secondly − and this is the interesting point − forwards the query to all the servents it is connected to. What happens at this step is that this way of widening the scope of queries risks to saturate the network thus compromising performance and scalability. In fact in August 2000, users reported that responses to their searches were fewer in number and slower to arrive than in the past: average Gnutella network traffic exceeded 10 query/sec per link which is the current 56Kb modem's extent [P2PAM]. Other P2P applications define different protocols in order to avoid problems like the one we have already mentioned, achieving remarkable results.

Freenet is an adaptive P2P application that permits the publication, replication and retrieval of data while protecting users' anonymity.

For our purposes it is interesting only showing how Freenet tries to avoid saturating the network when performing searches, still granting good results. The key issue is that queries are not routed aimlessly, but just towards peers known of being a potentially good target. Each request message contains a key to be searched, and each node locally has a "routing table" whose entries are hash values computed over known keys (the keywords). These entries are coupled with addresses of peers storing resources matching with the given keyword. This set of information is constantly updated during a node's lifetime, so refining the search mechanism with the elapsing of the time and the growing of the degree of interaction with the other peers. Figure 3.1 shows how this happens in Freenet.

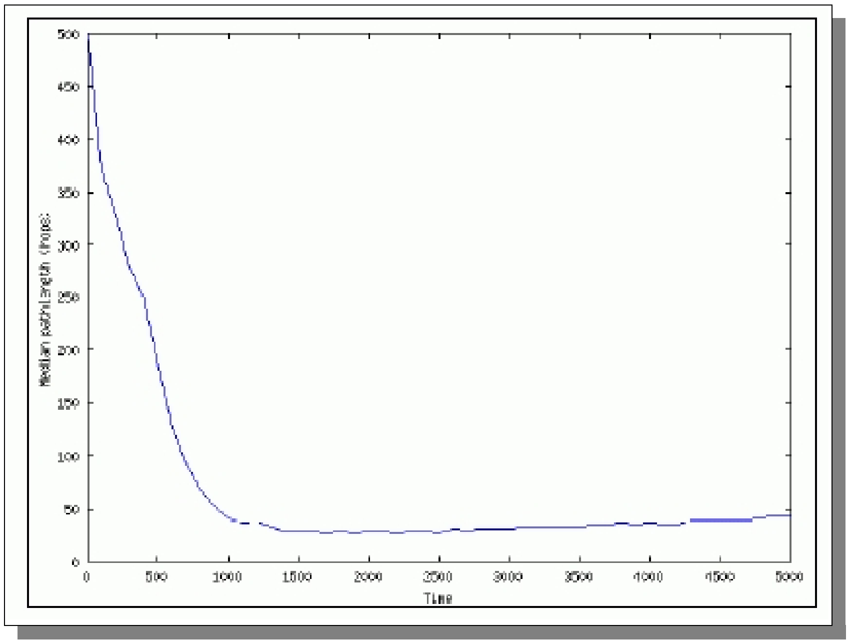


Figure 3.1 Search−Path length changing in better over time [P2PAM].

3.5.2 Case Study 2: SETI@home − Massively Distributed Computing for SETI

The SETI@home project is managed by a group of researchers at the Space Sciences Laboratory of the University of California, Berkeley. Its main attempt is to use distributed computing to perform a sensitive search for radio signals from extraterrestrial civilizations.

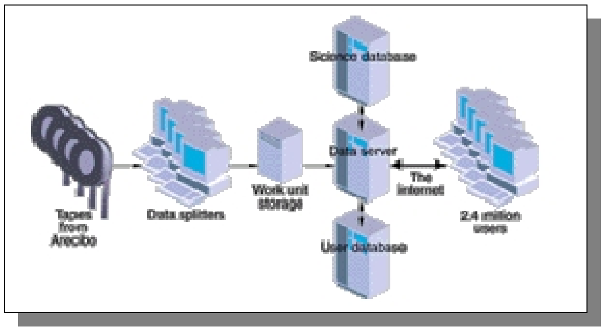


Figure 3.2 Cluster of servers for the SETI@home Project [SATH]

It could be interesting to analyze this project since it does not represent the classical peer−to−peer application, and it can even show us how valuable is the amount of resources known as the “dark matter” of the Internet. The SETI@home works as follows. All the radio signals are collected by the National Astronomy and Ionospheric Center’s telescope at Arecibo, Puerto Rico, and recorded continuously onto 35−Gbyte DLT tapes using 2−bit complex samples. The recorded tapes are shipped to Berkeley and subdivided in small work units. Each unit is then transferred on temporary storage for distribution to users. Once a “servant” connects to the “server”, it will be provided with one unit of work to be processed. All the processing will be potentially carried out off line, since no interaction is needed until the work has been accomplished and another unit will be downloaded. The main server is a cluster of three Sun™ Enterprise 450 series computers. The first one holds the user database, more than 2.8 million volunteers at the moment, the second machine holds the science database (sky coordinates, frequencies etc.) in an ever expanding array of redundant disks. The third server is in charge of distributing the work units [SATH]. As of 30 March 2001, 2.895.449 volunteers had run the software, donating a total of 611.327 years of CPU time for a total 7.793060 x 10^{20} flop.
At the best of our knowledge SETI@home is the largest distributed computation project in the world and it may be considered as the largest supercomputer ever existed as well.

3.6 Conclusions

Since distributed computing has become a widespread reality, researchers’ efforts have been directed towards the discovery of new computational paradigms able to improve distributed systems’ reliability and performance. In fact switching from a centralized solution to a purely distributed one is not enough. There are two major issues to keep in mind when designing a distributed system, namely: its architecture and the way distributed processes interact with one another. The most noticeable architectural patterns experienced in the years are named *ring*, *hierarchical* and *decentralized*. peer−to−peer systems are rooted in *decentralized* distributed systems, but besides mere architectural issues, their distinguishing characteristic resides in the role each node plays: the peer−to−peer philosophy foresees the existence of nodes all exposing the very same set of capabilities. What this means is that each node can act both as client and server. With this approach both fault tolerance and scalability turn out to be improved: having multiple nodes playing the same role, if some of them crashed, the same service would be provided by the remaining nodes. Not having anymore a single node in charge of providing the whole service even avoids bottlenecks, thus overcoming the scalability extents today's hierarchical systems may expose. Even though topology is important, today's P2P systems have shown how protocols are fundamental to the concept of scalability: the prime Gnutella's approach to requests propagation caused the saturation of the GnutellaNetwork in August 2000, so undermining the scalability expectations of the system. Anyway P2P systems are thought as being really good promising nowadays, and in order to have an idea of the impact these systems are supposed to have in the forthcoming years, it is interesting to take a look at what **Gartner**Consulting™ thinks about.

Given e−business trends, market drivers and the importance of partnership strategies, Gartner expects that P2P content network will become prevalent within the next five years. Gartner believes that by the year 2003, 30% of corporations will have experimented with Data Centered P2P applications for content distribution (0.7 probability). Gartner also believes that half of the current server−based content management vendors will add Data Centered P2P functionality to their product offerings by 2005 (0.7 probability)[GC].

4 Project JXTA

4.1 Introduction

In the former chapter the P2P paradigm for distributed computing has been introduced, exposing both its advantages and disadvantages. The whole discussion has been completely carried out on purely theoretical basis except the two case studies presented at the end of the chapter. But, what about real P2P computing? Is there any framework meant for peer−to−peer applications development, or developers should have to rewrite software components every time they need? It is well know that software reuse is a fundamental subject not only at enterprise level, since it let us save time by simply using software which has been both already written and tested by a potentially great number of users, and this means safer software along with time saving. Furthermore, another not trivial issue in today’s P2P projects and environments is the absence of a fundamental property such as interoperability. Applications such Gnutella or Freenet, defines two disjoint enclaves in the Internet, that are not capable to cooperate. Services offered by them are often similar, and in some cases overlapping; this implies unnecessary effort dispersion by developers. In conclusion, modern P2P applications address only a single function, run primarily on a limited set of platforms, and are unable to directly share data with other similar applications. The project JXTA, started at the beginning of the 2001, is aimed at overcoming these problems. The JXTA Project has been promoted by Sun Microsystems, Inc. before becoming an open source project distributed under an Apache Software Foundation like licence. But why the name “JXTA”? As stated in the previous chapter, the Internet has evolved mainly in a hierarchical way in the last two decades, until peer−to−peer computing has appeared.

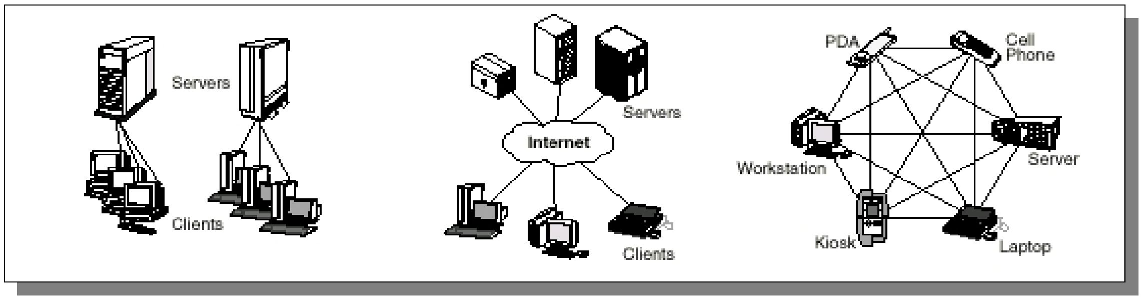


Figure 4.1 Distributed Computing Evolution [PJOIC]

So we can figure out peer−to−peer computing as being *juxtaposed* to the hierarchical client−server model of the Internet, hence the name *Project JXTA.* By supporting applications that are collaborative and communication oriented, Internet use can be more natural, intuitive and productive [PJOIC].

4.2 Project JXTA Objectives

The objectives of the Project JXTA are directly derived from the shortcomings and inefficiencies examined above. In particular the Project JXTA aims to address properties like *Interoperability, Platform Independence* and *Ubiquity* [PJTO].

- **Interoperability**. This is one of the most important issues concerning Project JXTA. The JXTA technology is designed to enable peers to discover each other and cooperate to give life to peer groups, i.e. communities of peers having some common interest. Peers might be part of different P2P systems or communities. Interoperability is provided defining JXTA as a set of protocols free to be implemented using whichever platform (language, operating system and hardware) developers may choose.
- **Platform Independence**. JXTA is designed to be independent not only from the programming language and the system platform used, but also to be

capable to adapt to different networking platforms. Different transport layers may be adopted, by implementing and using distinct “platform bindings”. A *platform binding* is an implementation of a set of the JXTA protocols (not necessary all of them), aimed for the desired platform.

- **Ubiquity**. JXTA technology is designed to be implemented on every device with a digital heartbeat, from sensors to PDA, from routers to desktop computers and servers and so on.

Before going deeper into the details of how these objectives are satisfied by the Project JXTA, it might be interesting to take a look at the architecture of the platform (from now on the word “platform” is used for referring to the the Project JXTA).

4.3 Project JXTA Architecture

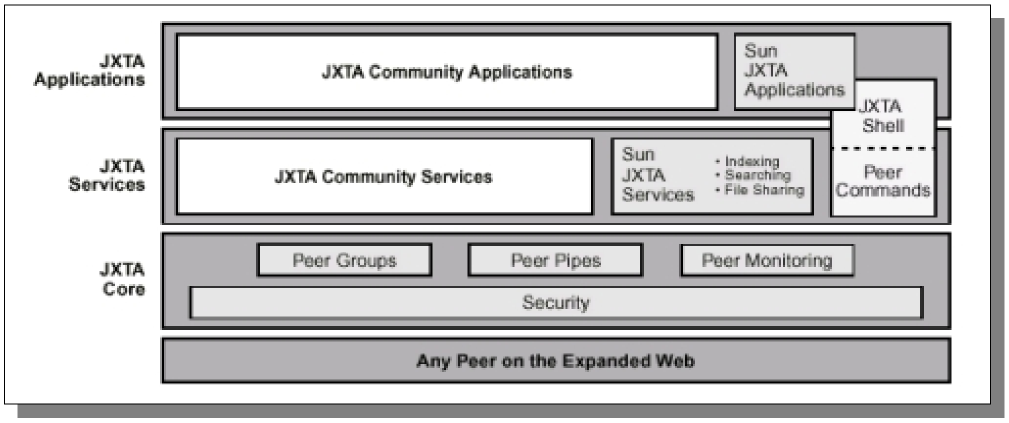


Figure 4.2 Project JXTA three layers architecture [PJJFPG]

While examining the different peer−to−peer systems in use today, the JXTA team found out a common layering structure. A classical peer to peer system can so be broken down into three main layers, namely: the *Core layer,* the

Service Layer, and the *Application Layer.* JXTA itself has been designed in order to reflect this structure with particular attention to each layer’s content (see fig. 4.2).

- **JXTA Core***.* At the bottom layer there is the core of the platform. This level deals with all the basic components a generic peer−to−peer system needs, such as peer establishment, group management, communication primitives, discovering mechanism, routing with firewall and NAT handling, and basic security services.
- **JXTA Services***.* This layer contains all the components which can be thought as not essential to every peer−to−peer system, or whose definition should be left up to developers. For example, services such as file or resource sharing, indexing and searching, are not necessary to every peer−to−peer system, and so they should not be placed within the core of the platform but in this middle−level layer instead.
- **JXTA Applications**. This layer encloses all the high level applications developers could devise using the set of services provided by the two lower layers. It is important to point out how JXTA applications are allowed to bypass the middle layer (the JXTA services layer) interacting directly with the core. Again this possibility highlights that components located in the middle layer are not needed by all peer−to−peer system implementation.

JXTA services can be further classified in *Peer Services, Peer Group Services* and *JXTA−Enabled Services.*

- **Peer Group Services***.* Each peer group defines a set of services that have to be implemented and supported by every peer belonging to the group or wishing to be part of it.
- **Peer Services***.* Unlike peer−group services, peer services may be implemented by each single peer independently from other peers or groups.

- **JXTA−Enabled Services**. These are services accessible only using *JXTA Pipes.* A JXTA pipe is the platform's specific communication primitive.

Besides JXTA−Enabled Services it is also possible to define and implement services accessible using different schemes (RMI, SOAP, XML−RPC, TCP/IP, etc.) in order to meet users' specific needs.

4.4 Project JXTA main components

In order to simplify the understanding of the following sections, we define and describe some of the fundamental concepts upon which the whole platform is based.

4.4.1 Peers

Peers are networked devices implementing a set of the JXTA Core protocols. Each peer is characterized by a given peer ID, and a set of network interfaces called *peer endpoints*. A peer ID along with its set of associated peer endpoints, uniquely identifies a peer in the JXTA network. It is not necessary for peers to be directly connected with one another to communicate, since intermediary peers can be used for routing purposes. Peers can be classified according to the basic tasks they are able to accomplish in the JXTA network [PJJFPG]:

- **Minimal Peer**. Such a peer is only able to exchange messages with other peers in the network, but it can not be used for message routing or for advertisement caching on behalf of other nodes. The existence of this class of peers is meant for letting simple and small devices such as today's PDAs be members of the JXTA network.

- **Simple Peer**. A simple peer is required to be able to interact with other nodes exchanging messages and to locally cache advertisements. That is why simple peers can answer to discovery requests issued by remote peers but they can neither perform routing operations nor propagate to other peers requests they receive.
- **Rendezvous Peer**. A rendezvous peer is a simple peer with the ability to propagate the requests it receives to other peers in the network, thus widening each request's scope. To every discovery request is assigned a TTL value (Time To Live), which will limit the propagation of the request itself. Every peer can be pre−configured to be a rendezvous peer, or it can dynamically become so. In the same way peers can be pre−configured to use a given rendezvous peer or they can dynamically bind to newly discovered rendezvous nodes. Rendezvous peer are fundamental since they act as a gathering point where peers can find each other and enlarge their visibility of the network.
- **Relay Peer**. Relay peers are in charge of acting as routers on behalf of peers who are not able to directly reach other nodes in the network, both due to the existence of firewalls or NATs and for the lack of routing information at the source. When peer A wants to send a message to peer B, it firstly looks into its cache for routing information; if nothing were found it would leverage on its relay peer asking it for routing information.

More details about rendezvous and relay peers are provided in section 4.6.

4.4.2 Peer Groups

Peer Groups are sets of peers all implementing a common set of services, the *Peer Group Services*. Each peer group is given a unique identifier, a *Peer Group ID*. Groups are arranged in a hierarchical way, with each group having a single parent. Peer groups have to implement a set of core services useful for granting peers some basic functionality.

- **Discovery Service***.* This service is used by peers in order to find out resources of any kind: a resource can be a peer, a peer group, a structured document, or whatever.
- **Membership Service***.* Once a peer has discovered a new group and wishes to join it, the *membership service* will manage this task accepting or refusing the applying peer according to the implemented membership service's policy.
- **Access Service***.* Peers belonging to the same group can interact with each other through higher level services. The access service is in charge of verifying one peer's right for making requests to another one before accepting to accomplish the requested task.
- **Pipe Service***.* This is one of the most important services since it deals with the standard communication means provided by the JXTA Technology.
- **Resolver Service***.* This service is used to send generic query requests to other peers. It can be extended in order to meet developer's specific needs.
- **Monitoring Service***.* This service is used to monitor peers' state.

It is important to recall that it is not necessary for a peer group to implement all the above services, but only the ones really needed.

4.4.3 JXTA Pipes

JXTA pipes are virtual communication channels used by peers in order to interact with each other. Peers do not need to be directly connected for exchanging messages, since intermediary nodes may be used for routing messages even through firewalls. Pipes have been designed to be asynchronous and unidirectional since they are meant to be the starting point for building more sophisticated communication mechanisms. Each pipe consists of two *pipe endpoints*, which are referred to as *input pipe* and *output pipe*. Informally a pipe endpoint corresponds to a peer's network interface that can be used for sending and/or receiving messages.

We have three types of pipes in the current platform implementation:

- **Point−to−Point Pipe**. This pipe is a unidirectional asynchronous communication channel connecting only two peers.
- **Propagate Pipe**. A propagate pipe is still unidirectional and asynchronous but implements a one−to−many communication pattern.
- **Secure Unicast Pipe**. A secure pipe is implemented upon TLS (Transport Layer Security) Version 1 in order to guarantee reliable private connections.

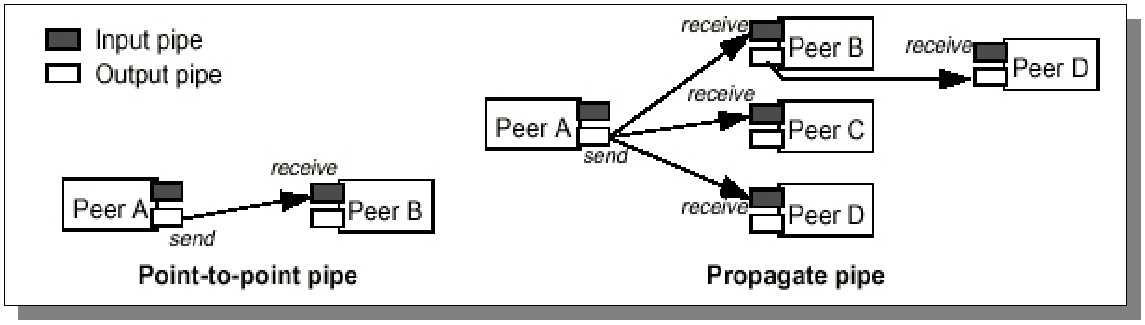


Figure 4.3 Examples of point to point and propagate JXTA pipes [PJJFPG].

4.4.4 Advertisements

As stated before, peers and peer groups need a way for publishing their own existence in the JXTA network as well as to advertise the presence of the resources they wish to grant to other nodes belonging to their same group. Even in order to join a given group, a generic peer should be enabled to first find it out. That is why every resource in the JXTA network is represented through an advertisement. In pursuit of interoperability, heterogeneous resources are advertised using language neutral meta−data structures, represented by XML documents. XML guarantees interoperability since today on every platform are available libraries for manipulating XML documents. When publishing each advertisement a default life time is given to it, if none is specified. Once the advertisement's life time expires, it can be republished for making the resource still available to other peers. The platform defines nine basic advertisements:

- **Peer Advertisement**. This document is aimed at advertising the existence of a peer in the JXTA network. A peer advertisement holds configuration information about a peer along with its name, its own JXTAID, its group JXTAID, a logical description of the peer and the set of implemented services.

```
<xs:complexType name="PA">
        <xs:element name="name" type="xs:string" minOccurs="0"/>
        <xs:element name="PID" type="JXTAID"/>
        <xs:element name="GID" type="JXTAID"/>
        <xs:element name="Desc" type="xs:anyType" minOccurs="0"/>
        <xs:element name="Dbg" type="xs:token" minOccurs="0"/>
        <xs:element name="Svc" type="jxta:serviceParams" minOccurs="0"
                                               maxOccurs="unbounded"/>
</xs:complexType>
<xs:simpleType name="JXTAID">
        <xs:restriction base="xs:anyURI">
        <pattern value="([uU][rR][nN]:[jJ][xX][tT][aA]:)+\-+"/>
        </xs:restriction>
</xs:simpleType>
<xs:complexType name="serviceParam">
        <xs:element name="MCID" type="JXTAID"/>
        <xs:element name="Parm" type="xs:anyType"/>
</xs:complexType>
```

Figure 4.4 Peer Advertisement Schema [PJJFPG]

- **Peer Group Advertisement**. This document advertises the existence of a peer group. It holds the following information: the group ID (a JXTAID), the module specification ID (MSID, a JXTAID identifying the module responsible for the peer group implementation), the logical name of the peer group, its logical description, and a potentially empty set of implemented services. Given the MSID a peer might locate a Module Implementation Advertisement or a Module Specification Advertisement, documents that descibe one of the potentially multiple implementations and the network behavior of the same service, respectively (see the following sections for further details).

```
<xs:complexType name="PGA">
        <xs:element name="GID" type="JXTAID"/>
        <xs:element name="MSID" type="JXTAID"/>
        <xs:element name="name" type="xs:string" minOccurs="0"/>
        <xs:element name="Desc" type="xs:anyType" min0ccurs="0"/>
        <xs:element name="Svc" type="jxta:serviceParam" minOccurs="0"
         maxOccurs="unbounded"/>
</xs:complexType>
```

Figure 4.5 Peer Group Advertisement Schema [PJJFPG]

- **Module Class Advertisement**. A module class advertisement is only meant for providing a description of what a given Module Class ID stands for, and could be useful for humans willing to create different implementations of an abstract service. The JXTA platform does not require to create and publish such an advertisement.

```
<xs:complexType name="MCA">
        <xs:element name="MCID" type="JXTAID"/>
        <xs:element name="name" type="xs:string" minOccurs="0"/>
        <xs:element name="Desc" type="xs:anyType" min0ccurs="0"/>
</xs:complexType>
```

Figure 4.6 Module Class Advertisement Schema [PJJFPG]

- **Module Specification Advertisement**. A module specification advertisement is used for clarifying what a given MSID stands for and which is the network behavior of a service. In case that multiple different implementations of the same service had to be interoperable, they should have the same module specification ID (MSID). A MSA is also used to define how the service should be invoked: simply using the service's API (locally), through a JXTA pipe, or through a proxy. All these information are encapsulated in the module specification advertisement (MSA).

```
<xs:complexType name="MSA">
        <xs:element name="MSID" type="JXTAID"/>
        <xs:element name="name" type="xs:string" minOccurs="0"/>
        <xs:element name="Crtr" type="xs:string" minOccurs="0"/>
        <xs:element name="SURI" type="xs:anyURI" min0ccurs="0"/>
        <xs:element name="Vers" type="xs:string"/>
        <xs:element name="Desc" type="xs:anyType" min0ccurs="0"/>
        <xs:element name="Parm" type="xs:anyType" min0ccurs="0"/>
        <xs:elemen  name="PipeAdvertisement"
                    type="jxta:PipeAdvertisement" min0ccurs="0"/>
        <xs:element name="Proxy" type="xs:anyURI" min0ccurs="0"/>
        <xs:element name="Auth" type="JXTAID" min0ccurs="0"/>
</xs:complexType>
```

Figure 4.7 Module Specification Advertisement Schema [PJJFPG]

- **Module Implementation Advertisement**. Since a service defined by a MSA might actually have different implementations, the concept of Module Implementation Advertisement (MIA) is provided for describing one of those particular implementation. It contains information such as the logical name of the service, the MSID it refers to, a compatibility tag, a tag identifying the code in charge of providing the service (i.e. in the Java implementation this will be a valid class name), an URI for downloading the implementation code, a tag stating who the service provider is and a set of arbitrary parameters.

```
<xs:complexType name="MIA">
        <xs:element name="MSID" type="JXTAID"/>
        <xs:element name="Comp" type="xs:anyType"/>
        <xs:element name="Code" type="xs:anyType"/>
        <xs:element name="PURI" type="xs:anyURI" min0ccurs="0"/>
        <xs:element name="Prov" type="string" min0ccurs="0"/>
        <xs:element name="Desc" type="xs:anyType" min0ccurs="0"/>
        <xs:element name="Parm" type="xs:anyType" min0ccurs="0"/>
</xs:complexType>
```

Figure 4.8 Module Implementation Advertisement Schema [PJJFPG]

- **Pipe Advertisement**. A pipe advertisement is an XML document describing a JXTA pipe. It is used by peers in order to dynamically bind their own endpoints before starting to exchange messages. The pipe advertisement points out the logical name of the pipe, its type (point to point, propagate, secure) and its own JXTAID.

```
<xs:complexType name="PA">
        <xs:element name="Id" type="JXTAID"/>
        <xs:element name="Type" type="string"/>
        <xs:element name="Name" type="string"/>
</xs:complexType>
```

Figure 4.9 Pipe Advertisement Schema [PJJFPG]

4.5 JXTA IDs

In the former section, JXTA identifiers have been mentioned without giving a strict definition of what they really are. JXTA identifiers are represented as URNs (Uniform Resource Names). URNs are a special form of URI "... intended to serve as persistent, location−independent resource identifiers". JXTA identifiers are used in order to univocally refer to resources in the JXTA network, so they identify peers, peer groups, pipes, module classes, module specifications and contents. A generic JXTAID URN representation has two main components:

- **The JXTA Name Space**. This field is equal to the string `jxta` and every JXTAID must belong to this name space.
- **The JXTAID**. This component can be even further divided into two sub−fields: the JXTAID format, and the unique ID.
  - **The JXTA Format**. JXTA identifiers have been designed to allow the usage of already existing identifications schemes. That is why different JXTA IDs formats are necessary. According to a given format it can be argued how the ID has been generated and extra information could be extracted from it.
  - **The JXTA Unique ID**. This is a string uniquely identifying the JXTA resource.

In figure 4.10 there is the JXTAID format specification based on the ABNF syntax [IETF RFC 2234].

```
<JXTAURN>     ::= "urn:" <JXTANS> ":" <JXTAIDVAL>

<JXTANS>      ::= "jxta"

<JXTAIDVAL>   ::= <JXTAFMT> "-" <JXTAIDUNIQ>

<JXTAFMT>     ::= 1 * <URN chars>

<JXTAIDUNIQ>  ::= 1 * <URN chars>

<URN chars>   ::= <trans> | "%" <hex> <hex>

<trans>       ::= <upper> | <lower> | <number> | <other> |
                  <reserved>

<upper>       ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" |
                  "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" |
                  "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" |
                  "Y" | "Z"

<lower>       ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
                  "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
                  "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
                  "y" | "z"

<hex>         ::= <number> | "A" | "B" | "C" | "D" | "E" | "F" |
                  "a" | "b" | "c" | "d" | "e" | "f"

<number>      ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
                  "8" | "9"

<other>       ::= "(" | ")" | "+" | "," | "-" | "." |
                  ":" | "=" | "@" | ";" | "$" |
                  "_" | "!" | "*" | "'"

<reserved>    ::= "%" | "/" | "?" | "#"
```

Figure 4.10 JXTAIDs syntax [PJJFPG]

4.5.1 JXTA IDs properties

Every JXTA ID, regardless of format or type, has the following properties:

- **Unambiguous**. It must be a complete reference to the resource.
- **Relatively Unique**. It must refer to a single resource.
- **Canonical**. References to the same resource should encode the same JXTA ID. This enables the comparing of IDs to determine if they refer to the same resource or not.
- **Opacity**. In their URN presentation JXTA IDs are assumed to be opaque. The

context of an ID within a protocol message generally is sufficient to establish its type. A JXTA binding may be able to interpret an ID if it supports the ID Format. Generally, only the immediate participants in a JXTA protocol need to understand the contents of a JXTA ID.

4.6 JXTA Protocols

As stated at the beginning of this chapter, JXTA achieves interoperability and platform independence merely defining itself as a set of protocols, and using XML for advertising resources. The six protocols building the JXTA core platform are:

- *Peer Discovery Protocol (PDP).*
- *Peer Resolver Protocol (PRP).*
- *Peer Information Protocol (PIP).*
- *Pipe Binding Protocol (PBP).*
- *Endpoint Routing Protocol (ERP).*
- *Rendezvous Protocol (RVP).*

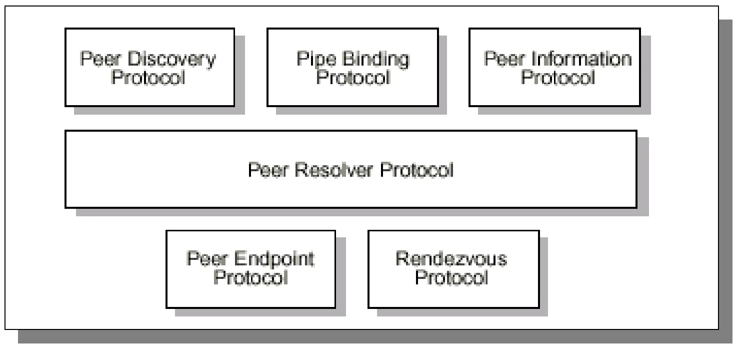


Figure 4.11 JXTA protocols layering [PJJFPG]

Peer Resolver Protocol. This is one of the most important protocols in the whole platform, since it represents the layer over which PDP, PIP and PBP are based. The peer resolver protocol defines a generic query/response mechanism suitable for building higher level specific protocols. In order to define other protocols based upon the PRP it is necessary to define two new request and response schema starting from the ones depicted in figures 4.12 and 4.13.

```
<xs:element name="ResolverQuery" type="jxta:ResolverQuery"/>

<xs:complexType name="ResolverQuery">
    <xs:element name="Credential" type="xs:anyType" minOccurs="0"/>
    <xs:element name="SrcPeerID" type="xs:anyURI"/>
    <!-- This could be extended with a pattern restriction -->
    <xs:element name="HandlerName" type="xs:string"/>
    <xs:element name="QueryID" type="xs:string"/>
    <xs:element name="Query" type="xs:anyType"/>
</xs:complexType>
```

Figure 4.12 Peer Resolver Query Schema [PJJFPG]

```
<xs:element name="ResolverResponse" type="ResolverResponse"/>
<xs:complexType name="ResolverResponse">
    <xs:element name="Credential" type="xs:anyType" minOccurs="0"/>
    <xs:element name="HandlerName" type="xs:string"/>
    <xs:element name="QueryID" type="xs:string"/>
    <xs:element name="Response" type="xs:anyType"/>
</xs:complexType>
```

Figure 4.13 Peer Resolver Response Schema [PJJFPG]

Peer Discovery Protocol. This is the standard protocol used by peers for discovering resources scattered around the JXTA network. It is based on the PRP and so defines both a request message format and a reply message format. Rendezvous peers are fundamental to this protocol, since they are in charge of forwarding requests to other known peers. Figure 4.11 depicts a potential scenario where peer A only knows about the existence of the rendezvous peer R1, and so sends its queries to it. Then R1 will forward the same requests both to other simple peers and to other known rendezvous peers (remember that simple peers are not allowed to forward incoming requests, they can just try to satisfy them).

A TTL and a maximum amount of desired replies is assigned to each request. The service is classified as being a *best effort service*: nothing is granted a priori.

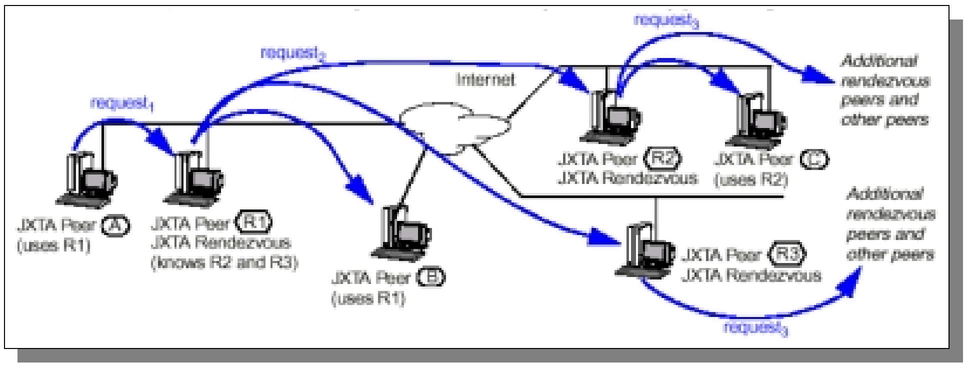

request$_1$
request$_2$
request$_3$
Internet
JXTA Peer A (uses R1)
JXTA Peer R1 JXTA Rendezvous (knows R2 and R3)
JXTA Peer B (uses R1)
JXTA Peer R2 JXTA Rendezvous
JXTA Peer C (uses R2)
JXTA Peer R3 JXTA Rendezvous
Additional rendezvous peers and other peers
Additional rendezvous peers and other peers
request$_3$

Figure 4.14 PDP example [PJJFPG]

```
<xs:element name="DiscoveryQuery" type="jxta:DiscoveryQuery"/>
<xs:complexType name="DiscoveryQuery">
      <!-- this should be an enumeration -->
      <xs:element name="Type" type="xs:string"/>
      <xs:element name="Threshold" type="xs:unsignedInt"
                                          minOccurs="0"/>
      <xs:element name="PeerAdv" type="jxta:PA" minOccurs="0"/>
      <xs:element name="Attr" type="xs:string" minOccurs="0"/>
      <xs:element name="Value" type="xs:string" minOccurs="0"/>
</xs:complexType>
```

Figure 4.15 Discovery Query Message schema [PJJFPG]

```
<xs:element name="DiscoveryResponse" type="jxta:DiscoveryResponse"/>
<xs:complexType name="DiscoveryResponse">
      <!-- this should be an enumeration -->
      <xs:element name="Type" type="xs:string"/>
      <xs:element name="Count" type="xs:unsignedInt" minOccurs="0"/>
      <xs:element name="PeerAdv" type="xs:anyType" minOccurs="0">
            <xs:attribute name="Expiration" type="xs:unsignedLong"/>
      </xs:element>
      <xs:element name="Attr" type="xs:string" minOccurs="0"/>
      <xs:element name="Value" type="xs:string" minOccurs="0"/>
      <xs:elementname="Response" type="xs:anyType"
                                maxOccurs="unbounded">
            <xs:attribute name="Expiration" type="xs:unsignedLong"/>
      </xs:element>
</xs:complexType>
```

Figure 4.16 Discovery Response Message schema [PJJFPG]

Peer Information Protocol. Once a peer has been discovered through the PDP, it might be monitored using the peer information protocol. This is a PRP based query/reply protocol, which defines both a query message format and a reply message format.

```
<xs:element name="PeerInfoResponse" type="jxta:PeerInfoResponse"/>
<xs:complexType name="PeerInfoResponse">
     <xs:element name="sourcePid" type="xs:anyURI"/>
     <xs:element name="targetPid" type="xs:anyURI"/>
     <xs:element name="uptime" type="xs:unsignedLong" minOccurs="0"/>
     <xs:element name="timestamp" type="xs:unsignedLong"
      minOccurs="0"/>
     <xs:element name="response" type="xs:anyType" minOccurs="0"/>
     <xs:element name="traffic" type="jxta:piptraffic" minOccurs="0"/>
</xs:complexType>

<xs:complexType name="piptraffic">
      <xs:element name="lastIncomingMessageAt"  type="xs:unsignedLong
       minOccurs="0"/>
      <xs:element name="lastOutgoingMessageAt"  type="xs:unsignedLong"
      minOccurs="0"/>
      <xs:element name="in" type="jxta:piptrafficinfo" minOccurs="0"/>
      <xs:element name="out"  type="jxta:piptrafficinfo"
      minOccurs="0"/>
</xs:complexType>

<xs:complexType name="piptrafficinfo"
      <xs:element name="transport"ype="xs:unsignedLong"
      maxOccurs="unbounded">
            <xs:attribute name="endptaddr" type="xs:anyURI"/>
      </xs:element>
</xs:complexType>
```

```
<xs:element name="PeerInfoQueryMessage"
            type="jxta:PeerInfoQueryMessage"/>
<xs:complexType name="PeerInfoQueryMessage">
      <xs:element name="sourcePid" type="xs:anyURI"/>
      <xs:element name="targetPid" type="xs:anyURI"/>
      <xs:element name="request" type="xs:anyType" minOccurs="0"/>
</xs:complexType>
```

Figure 4.17 Peer Information Protocol request and reply message schema [PJJFPG]

Pipe Binding Protocol. *JXTA pipes* are virtual communication channels which can be dynamically bound to a peer at runtime simply providing a pipe advertisement. Roughly speaking the *pipe binding protocol* is in charge of finding a *pipe endpoint* bound to the same pipe advertisement (this is an input−pipe). If such an endpoint was found, the connection would be established using the most suitable protocol available to both of the peers. The query message may ask only for *fresh* information (not read from cache), or it may specify a particular peer ID meaning that just that particular peer is allowed to answer.

Endpoint Routing Protocol. Since it may happen that a peer does not know all the necessary information for routing a message to the desired destination peer, or the destination peer might be behind a firewall or a NAT, the JXTA project envisages the existence of special peers able to handle all these scenarios: the *relay peers* (see former sections for details). Peers use the *endpoint routing protocol* in order to interact with relay peers. If a peer is not able to find a route to another node in the network, it queries its own relay peer for a possible path (just a sequence of hops). If the relay is not able to provide a reply with its local information, it forwards the request to other known relay peers, if available. Once the requesting peer has obtained the desired information it will send its message which will be routed following an adaptive source routing policy. In other words the source peer specifies the complete sequence of hops its message should follow, but should the network topology change in the meantime, the routing nodes would perform an attempt to dynamically find out a new available route. The whole process is limited by the TTL associated with the message. Even the endpoint routing protocol is classifiable as a *best effort service*. In figure 4.18 is depicted a possible interaction.

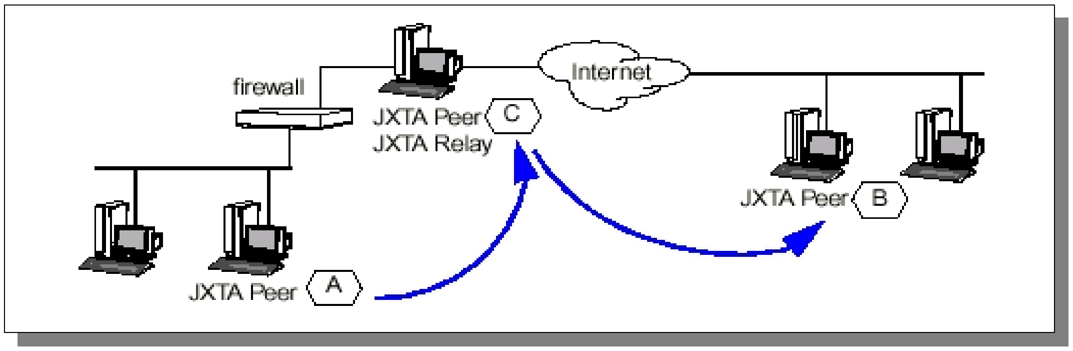


Figure 4.18 ERP example with a relay peer routing messages through a firewall [PJJFPG]

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:EndpointRouterQuery>
        <Credential> credential </Credential>
        <Dest> peer id of the destination </Dest>
        <Cached>
                true: if the reply can be a cached reply
                false: if the reply must not come from a cache
        </Cached>
</jxta:EndpointRouterQuery>
```

Figure 4.19 ERP route query message schema [PJJFPG]

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:EndpointRouterAnswer>
        <Credential> credential </Credential>
        <Dest> peer id of the destination </Dest>
        <RoutingPeer>
                Peer ID of the router that knows a route to DestPeer
        </RoutingPeer>
        <RoutingPeerAdv>
                Advertisement of the routing peer
        </RoutingPeerAdv>
        <Gateway> ordered sequence of gateway </Gateway>
        < ..................>
        <Gateway> ordered sequence of gateway </Gateway>
</EndpointRouterAnswer>
```

Figure 4.20 ERP route response message schema [PJJFPG]

Rendezvous Protocol*.* In the former examples we have seen how important is propagating messages in the JXTA network to involve as many peers as possible in a generic task. The rendezvous protocol is meant for this purpose: it is in charge of propagating messages through the network taking into account issues such as TTL handling, duplication, and loopback detection.

So what we can argue now is that all the core protocols are based on a request/reply pattern, connections among peers are not meant to be persistent, the default routing algorithm is inherently distributed and truly adaptive making it useful even in highly dynamic and ad−hoc networks. JXTA peers are not required to implement all the above protocols, but only the ones they really need, and if desired they can be replaced by other implementation specific protocols, for example to provide different qualities of service (*QoS*).

4.7 Conclusions

Project JXTA has been devised in order to cope with the absence of a network programming platform specifically designed for peer−to−peer applications development and deployment. Project JXTA main targets are:

- **Interoperability**. Any P2P system built with JXTA can talk to each other.
- **Platform Independence**. JXTA can be implemented with any programming language and run on any software and hardware platform.
- **Ubiquity**. JXTA can be deployed on any device with a digital heartbeat.

In pursuit of interoperability JXTA has been merely defined as a set of six protocols each of them meant for a particular usage: JXTA includes protocols for the discovering and publishing of resources, for monitoring peers and for routing issues. Fundamental to interoperability is the usage of XML: JXTA resources are published through XML documents which can later on be retrieved by other peers. As shown in figure 4.21, the Project JXTA protocols establish a virtual network on top of existing real networks, hiding their physical topologies.

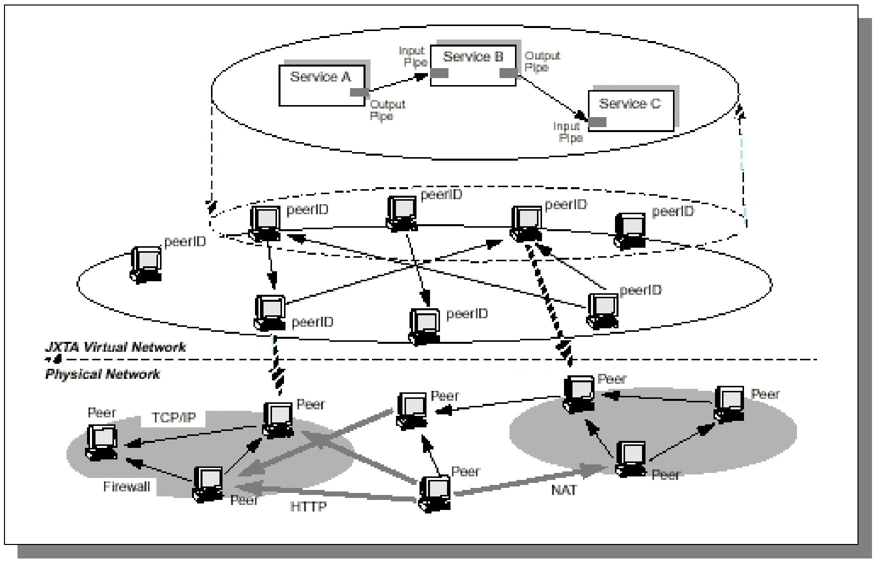


Figure 4.21 JXTA Virtual Network

5 The Anthill Project

5.1 Introduction

In the previous three chapters we have been investigating a set of concepts fundamental to the development of what this chapter introduces: the Anthill Project. Running over these topics, they were concerned with *complex adaptive systems*, *peer−to−peer computing* and the *JXTA Technology.* Complex adaptive systems, based on *autonomous agents*, are important due to their property of exposing *emergent behaviors:* the system, according to the steps performed by the interacting agents, will expose a behavior not foreseeable simply knowing the behavior of the system's basic components. The most relevant thing to notice it that there is no explicit coordination among agents, and that even few changes in the way they behave can lead to deeply different global behaviors. Our interest in the peer−to−peer distributed computing paradigm is due to the need to find out and adopt a purely decentralized solution for developing and deploying new distributed applications able to overcome the extents today's systems expose, gaining in performance and reliability. Then we have introduced the JXTA Technology since it can be thought as being the only framework meant for developing and deploying pure peer−to−peer systems guaranteeing interoperability and portability, nowadays. All these topics are brought together by a common idea: easily solve complex problems in a purely decentralized way. This is what the Anthill Project is about and aims to. Anthill is a framework for development, testing and deployment of peer−to−peer systems based on the *multi−agent* system (MAS) paradigm. Since one of the most important examples of complex adaptive systems based on MAS is the *Virtual Ants System* (see chapter 2), Anthill is inspired by concepts and ideas borrowed from such a system. An Anthill system can be thought as a collection of nodes, the *nests,* connected together in a network, the Anthill network. The autonomous agents travelling through the web of nests are called *ants*. An Anthill system is a peer−to−peer system that provides its users with a customizable set of services. Developers are allowed to implement an Anthill system for content sharing purposes, or for distributing heavily CPU bound computations among a wide set of machines or whatever. The Anthill Project follows a micro−kernel like approach: a set of basic services, such as communication mechanisms between nodes, discovering on new nests, storage management and so on, are provided by the Anthill infrastructure, while the real high−level services are implemented by ants. The service logic resides in the ants' algorithms, with the enormous advantage that implementing and deploying a new service is merely accomplished by devising new ant species that will be subsequently plugged into the Anthill platform. Along with the run−time environment, useful for executing real Anthill−based applications, an Anthill Project implementation comes with a simulation environment as well, whose API does not differ from the one exposed by the run−time environment, thus simplifying the process which goes from development to deployment. The Anthill Project goes even further, adding evolutionary computing techniques such as genetic algorithms. During off−line simulations ant species algorithms can be evaluated, compared, and even mixed in order to obtain new better ant species. For these purposes ants are characterized by a set of configuration parameters representing the ant's genetic code: mixing these parameters we can derive new ant species exposing completely new behaviours.

5.2 The Anthill Project's main components: Nests & Ant Species

5.2.1 Anthill Nests

As stated in the former section the Anthill key constituents are *nests* and *ant species*. An Anthill nest is a middleware in charge of granting a set of core services that ants will use while performing their tasks. These services range from generic storage management, to management of routing data structures on behalf of ants and communication support.

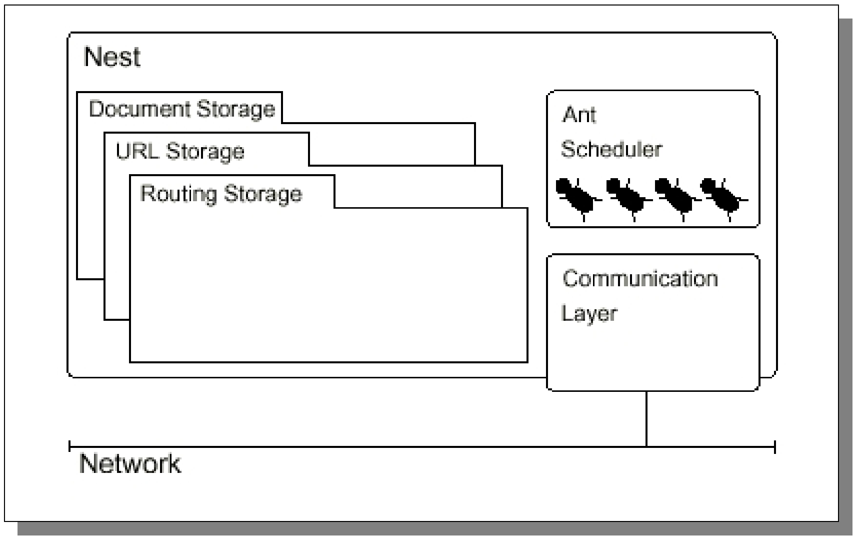


Figure 5.1 Anthill Nest's logical structure [BMM−09−01]

In figure 5.1 is depicted the logical structure of an Anthill nest. It is mainly composed by:

- *A set of Storages.*
- *An Ant Scheduler.*
- *A Communication Layer.*

Storage. Basically three types of storages are provided by an Anthill Nest implementation, namely: a *document storage*, an *URL storage*, a *routing storage.* A document storage is used for managing persistent resources, possibly shared among all the active nests in the Anthill network. It consequently contains potentially heterogeneous resources locally available to the Anthill nest. On the other side an URL storage handles a set of references to remotely stored resources. The most interesting component is the routing storage. These are data structures laid by ants when visiting an Anthill nest, that store routing information useful to other ants belonging to the same species. There are no constrains concerning what an ant−species routing data structure should be. This is possible since nests are only in charge of storing these resources, but are not required to be able to handle them: this is up to ants. This implies that there will be different routing policies for each ant species acting in the Anthill environment.

Ant Scheduler. Besides providing basic services involving communication and security, an Anthill nest has to let received ants execute their own tasks locally. In order to do this a nest has to provide an ant scheduler in charge of multiplexing local resources among the received ants according to some policy. The Anthill specification does not impose any specific way this task should be accomplished, living all the implementation details up to developers.

Communication Layer. Anthill nests are even responsible to grant ants a way for moving from one node to another through the Anthill network. This service, we refer to as the *Anthill GateService*, is the only compulsory network service every Anthill binding is required to provide. Other network services may then be added according to specific developers' needs.

Upon Anthill nests, are based high−level applications that represent the local interface between the nest's user and the P2P network [BMM−09−01]. High−

level applications interact with the Anthill infrastructure performing requests and listening for replies. Requests are handled by nest instances. What a nest does is selecting the most suitable ant species for satisfying the issued request. Then one or more ants are created and scheduled for local execution.

5.2.2 Anthill Ant Species

The Anthill Project lets developers implement high level services devising new *ant species*. Each ant species may be defined by an interface giving an high level description of what the ant is meant for. Subsequently, different implementations of the same service (ant species) are expected. Anthill compatible ant species are obviously required to use the Anthill API for interacting with the hosting nests. Recall that both the run−time environment and the simulation environment expose the same API. This API is used by ants for accessing resource storages, routing data structures, and network services provided by Anthill nests. Ants are executed locally to a nest in a sand box for security reasons. This way ants access to local resources can be controlled and limited as desired.

5.3 The Anthill Project Evolutionary Framework

As previously stated, the Anthill Project adopts evolutionary techniques for improving various characteristics of a P2P system [BMM−09−01]. Each Anthill ant species can be characterized by a unique chromosome that defines the ant species’ unique behavior in the Anthill environment. An ant species chromosome can be actually represented by a simple set of parameters that addict the ant’s algorithm execution. For example we can parameterize an ant algorithm introducing an exploration probability parameter. This parameter should influence the “willing” of the virtual ant to test new paths while travelling towards a resource, regardless of what is reputed to be the best path known in the network. Acting over this parameter different behaviors can be experienced, and the best ant can be selected according to some criteria. For example, if we were interested in the minimization of the total path length traversed by the ants while searching resources, we could test different ant species characterized by chromosomes differing in the *exploration parameter*, and observe which one better fits our requirements. These techniques could be applied even in the run−time environment, enabling nests to rate different ant species on the basis of a given criterion.

5.4 The Anthill Project Three Layers Structure

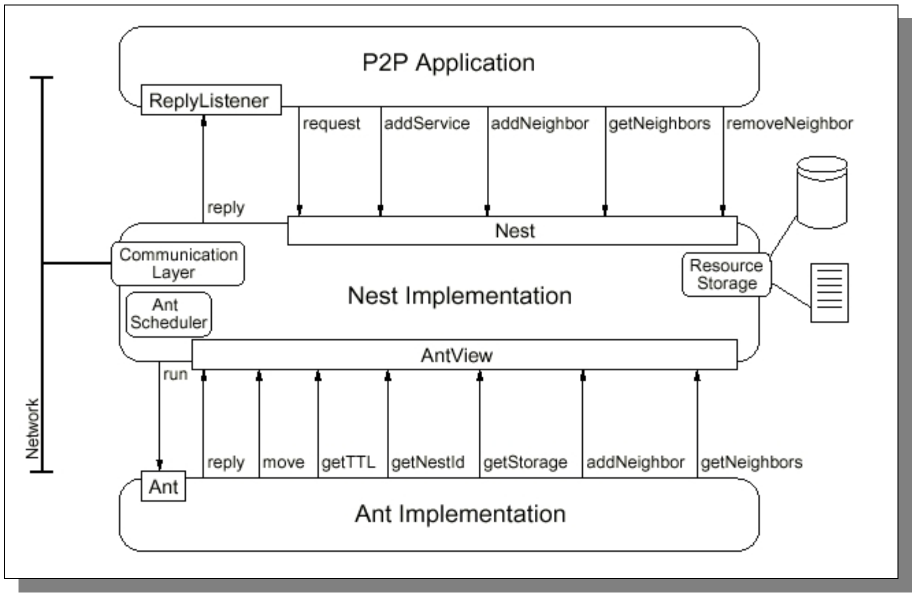


Figure 5.2 The Anthill Project Structure [BMM−09−01]

The Anthill Project is structured as depicted in Figure 5.2. In the middle there is the Anthill Project infrastructure, represented by the Anthill nest implementation.

On top of the Anthill infrastructure higher−level applications can be layered. These are mere interfaces between the nest’s users and the real P2P services implemented by ants. There should not be direct interaction between high−level applications and ants: nests have to be the means by which requests are mapped to ants and responses are returned to users.

Different *bindings*, or *implementations*, of the Anthill Project are admitted. The one proposed along with this thesis is based on the formerly introduced JXTA Technology. The following chapters respectively deal with the JXTAnthill Project architecture and the JXTAnthill Project implementation details. The last chapter exposes some preliminary simulation results related to an experimental ant species implementing a file−sharing service.

6 JXTAnthill: JXTA based Anthill binding

6.1 JXTAnthill

As stated in the previous chapter the Anthill Project defines a set of interfaces to be implemented in order to have a working “*binding*” of the project. This is an extremely useful and interesting approach since it let us devise different Anthill implementations each one based on the most appropriate technology. At the moment of writing this thesis the most interesting platform for developing peer−to−peer distributed systems surely was represented by the formerly introduced JXTA Project. That is due to the set of facilities it exposes: this leads to a rapid development of a P2P system, bereaving programmers of the burden of solving non−trivial problems such as peer and resource discovery, communication primitives, firewall handling and other related issues, in a purely decentralized environment. This justifies the choice of JXTA as the basis for our Anthill implementation. As previously seen, JXTA offers a set of services while not imposing developers to use them all. This is why the JXTAnthill implementation relies on a strict subset of the JXTA core services, namely: the *Peer Discovery Protocol* (PDP), the *Pipe Binding Protocol* (PBP) and the *Rendezvous Protocol* (RVP) [PJJFPG]. Before going deeper into the details concerning the integration of these protocols with the Anthill Project, it is necessary to take a look at the architectural design of JXTAnthill.

6.2 JXTAnthill: Project’s Architecture

The Anthill Project is composed by a set of hierarchically organized packages, namely:

- **anthill**. In this package are located sub packages and interfaces that should be implemented in order to have an Anthill Project binding.
- **ants**. This is the default package where ant species implementations should be located.
- **antsim**. This package contains the simulation environment implementation. This implementation has to be compliant with the specifications given by the set of interfaces located in the anthill package and in its sub packages.
- **jxtaimpl**. This is the JXTA based Anthill Project implementation proposed in this thesis. Obviously it has to be compliant with the interfaces defined in the packages named *anthill* and its sub−packages.

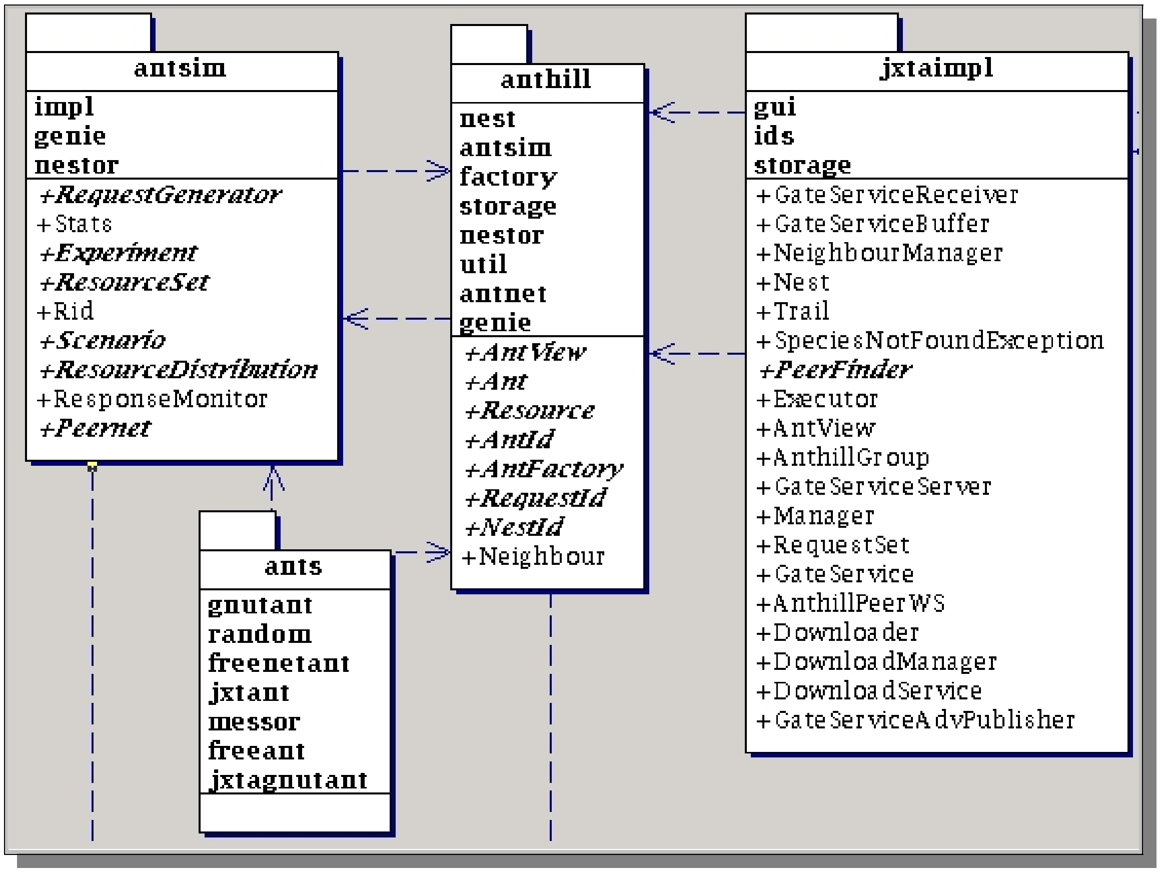


Figure 6.1 Detail of the Anthill Project first level UML Class Diagram

For our purposes, the only packages of interest are *anthill*, *jxtaimpl* and *ants*. Let us now analyse each of them separately.

6.3 Package *anthill*

As one can see in Figure 6.1, this package contains a set of first level interfaces plus a set of other sub packages as well. The interfaces are:

- **anthill.AntView**. This interface defines the only operations an ant can perform once reached a nest. For security concerns the ant is not allowed to directly interact with the hosting nest, but with an instance of a class implementing the *anthill.AntView* interface. This class will just map calls over the appropriate Nest implementation.
- **anthill.Ant**. This interface defines the default set of methods a new ant species has to provide.
- **anthill.Resource**. Anthill resource implementations must implement this interface.
- **anthill.AntId**. Each ant travelling through the Anthill network has to be uniquely identifiable, and implementation identifiers must implement this interface.
- **anthill.AntFactory**. A classical *factory−based design pattern* is used in order to obtain ants as needed. So there must be a factory for each ant species implementation.
- **anthill.RequestId**. Requests have to be uniquely identifiable, and implementation identifiers have to be compliant with this interface.
- **anthill.NestId**. Each nest in the Anthill network is unique and uniquely identified by its own NestId implementation.

All these interfaces are fundamental to every implementation. The *anthill* package encloses many other sub packages, but the only ones relevant to our discussion are: *anthill.nest* and *anthill.storage*.

6.4 Package *anthill.nest*

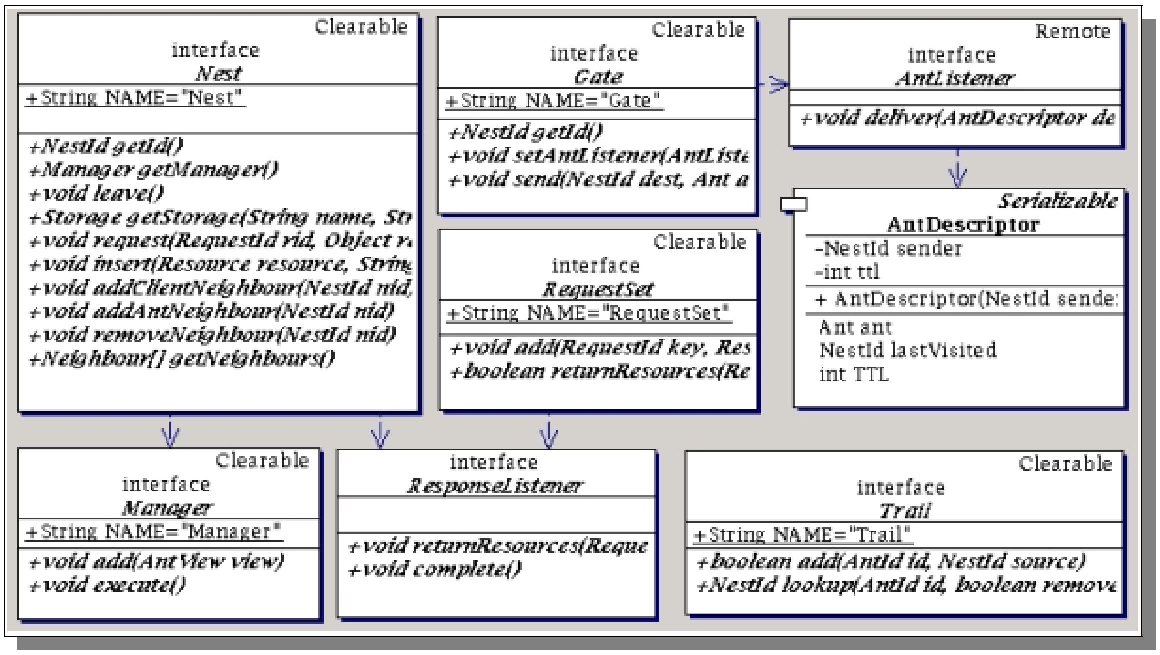

Clearable
interface
Nest
+String NAME="Nest"
+NestId getId()
+Manager getManager()
+void leave()
+Storage getStorage(String name, St
+void request(RequestId rid, Object r
+void insert(Resource resource, String
+void addClientNeighbour(NestId nid,
+void addAntNeighbour(NestId nid)
+void removeNeighbour(NestId nid)
+Neighbour[] getNeighbours()
Clearable
interface
Gate
+String NAME="Gate"
+NestId getId()
+void setAntListener(AntListe
+void send(NestId dest, Ant a
Remote
interface
AntListener
+void deliver(AntDescriptor de
Clearable
interface
RequestSet
+String NAME="RequestSet"
+void add(RequestId key, Res
+boolean returnResources(Re
Serializable
AntDescriptor
-NestId sender
-int ttl
+ AntDescriptor(NestId sende
Ant ant
NestId lastVisited
int TTL
Clearable
interface
Manager
+String NAME="Manager"
+void add(AntView view)
+void execute()
interface
ResponseListener
+void returnResources(Reque
+void complete()
Clearable
interface
Trail
+String NAME="Trail"
+boolean add(AntId id, NestId source)
+NestId lookup(AntId id, boolean remove

Figure 6.2 anthill.nest UML Class Diagram

The *anthill.nest* package contains the set of interfaces each nest implementation has to implement. Them all are noteworthy for the prosecution of our discussion:

- **anthill.nest.Nest***.* The *Nest* interface defines all the basic functions a generic Anthill nest should provide. They are mainly related to storage management, requests handling and neighbours management.
- **anthill.nest.Gate***.* The *Gate* interface has to be implemented by those classes in charge of handling all the communication related issues. It represents the means by which ants can move from one nest to another in the Anthill network.

- **anthill.nest.AntListener**. This interface has to be implemented by those classes willing to receive ants from remote nests.
- **anthill.nest.RequestSet**. This interface describes a table for mapping requests to listeners.
- **anthill.nest.ResponseListener**. This interface has to be implemented by those classes willing to receive responses to previously issued requests.
- **anthill.nest.Manager**. The *Manager* interface has to be implemented by classes in charge of scheduling and executing ants locally.
- **anthill.nest.Trail**. This interface should be used in order to implement classes useful for storing associations between ant identifiers and the nest they come from.

6.5 Package *anthill.storage*

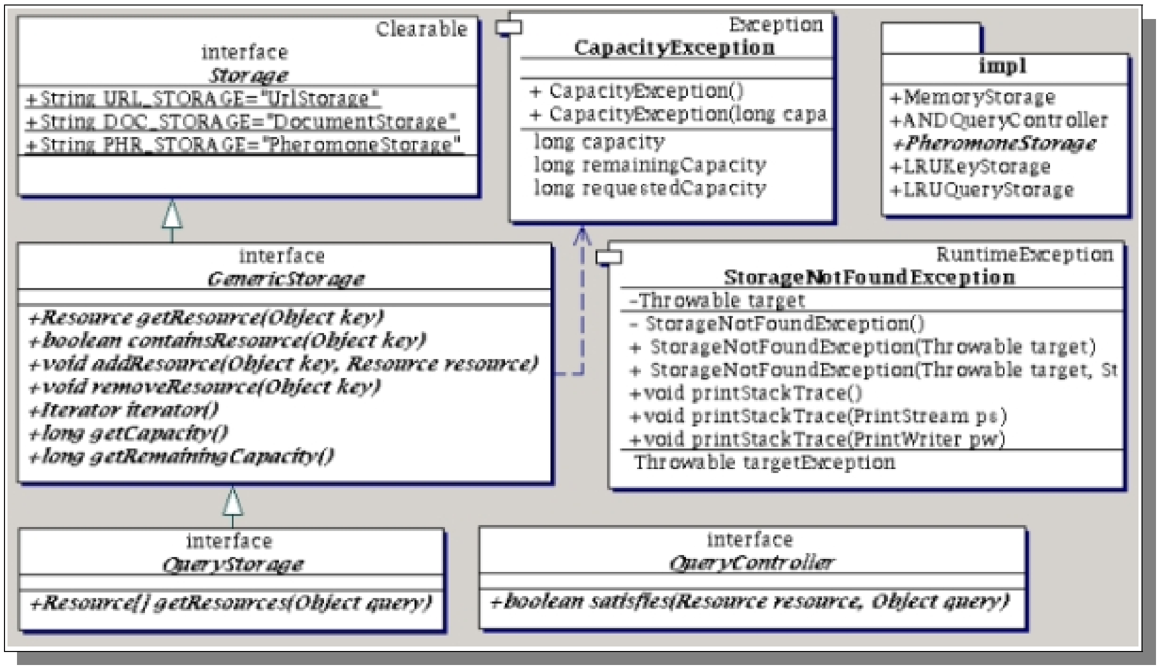

Clearable
interface
Storage
+String URL_STORAGE="UrlStorage"
+String DOC_STORAGE="DocumentStorage"
+String PHR_STORAGE="PheromoneStorage"
Exception
CapacityException
+ CapacityException()
+ CapacityException(long capa
long capacity
long remainingCapacity
long requestedCapacity
impl
+MemoryStorage
+ANDQueryController
+PheromoneStorage
+LRUKeyStorage
+LRUQueryStorage
interface
GenericStorage
+Resource getResource(Object key)
+boolean containsResource(Object key)
+void addResource(Object key, Resource resource)
+void removeResource(Object key)
+Iterator iterator()
+long getCapacity()
+long getRemainingCapacity()
RuntimeException
StorageNotFoundException
-Throwable target
- StorageNotFoundException()
+ StorageNotFoundException(Throwable target)
+ StorageNotFoundException(Throwable target, St
+void printStackTrace()
+void printStackTrace(PrintStream ps)
+void printStackTrace(PrintWriter pw)
Throwable targetException
interface
QueryStorage
+Resource[] getResources(Object query)
interface
QueryController
+boolean satisfies(Resource resource, Object query)

Figure 6.3 anthill.storage UML Class Diagram

What the anthill.storage package defines is a set of general purpose storage interfaces to be implemented by real applications. Three of these interfaces are hierarchically organized, and these are the ones we are interested in:

- **anthill.Storage***.* This is an empty interface that must be implemented by all storages. Furthermore it defines a set of basic identifiers that are used to distinguish among three basic kinds of storages. An *URL storage* should be used in order to store references to documents or resources scattered all around the Anthill network. A *document storage* is in charge of managing all the shared resources locally available to the nest. Last but not least there is the *pheromone storage* which could be used by nests in order to manage the pheromone data structures laid by hosted ants.
- **anthill.GenericStorage***.* This interface extends the former one by adding a set of facilities a storage implementation should provide. For example there should be methods meant for adding and removing resources or for collecting information about the storage usage.
- **anthill.QueryStorage***.* The *QueryStorage* interface even further extends the set of tasks a storage should implement by adding a method for obtaining more then a single resource per query.

Still noteworthy are the two exceptions located in this package, namely the *StorageNotFoundException* and the *CapacityException.* Since in an Anthill implementation a storage is associated with a given ant species, should an ant try to use a not already instantiated storage, a *StorageNotFoundException* is thrown. The *CapacityException* is related to the fact that a storage is limited in size for security reasons.

6.6 Package *jxtaimpl*

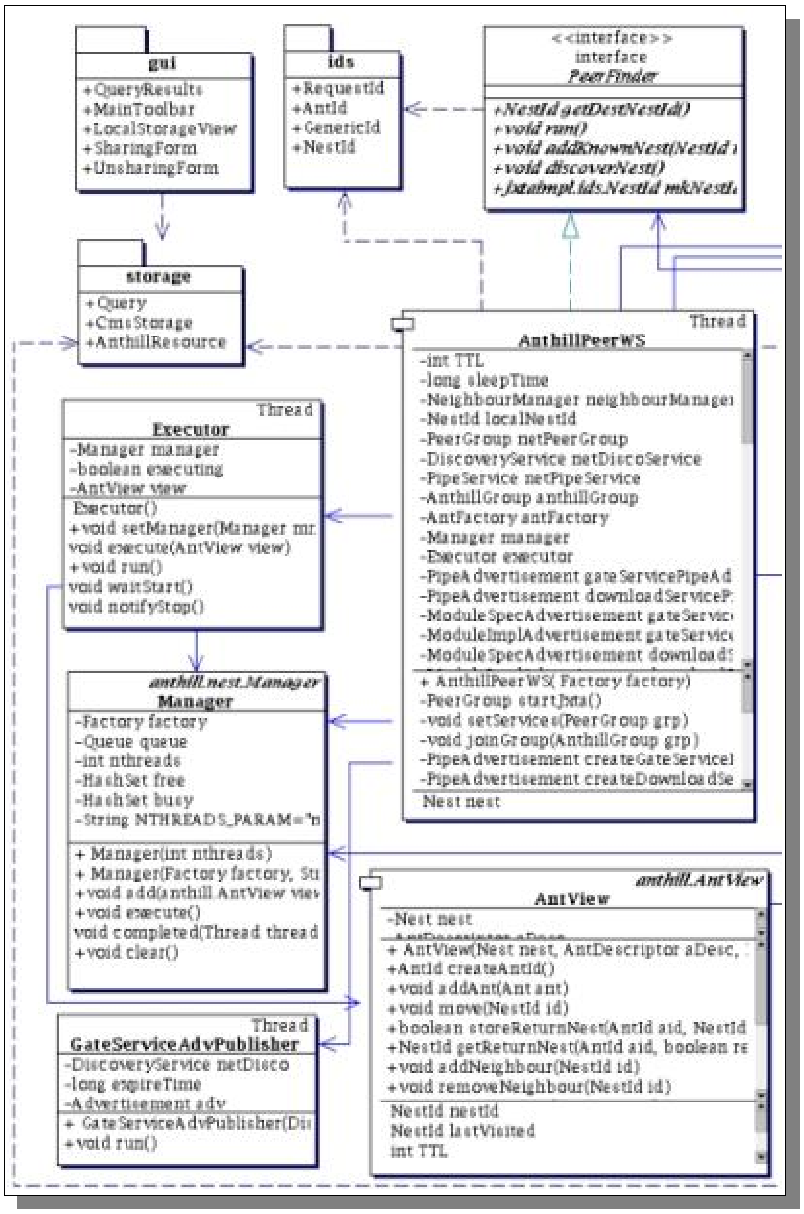

gui
+QueryResults
+MainToolbar
+LocalStorageView
+SharingForm
+UnsharingForm
ids
+RequestId
+AntId
+GenericId
+NestId
<<interface>>
interface
PeerFinder
+NestId getDestNestId()
+void run()
+void addKnownNest(NestId
+void discoverNest()
+jxtaimpl.ids.NestId mkNest
storage
+Query
+CmsStorage
+AnthillResource
Thread
AnthillPeerWS
-int TTL
-long sleepTime
-NeighbourManager neighbourManager
-NestId localNestId
-PeerGroup netPeerGroup
-DiscoveryService netDiscoService
-PipeService netPipeService
-AnthillGroup anthillGroup
-AntFactory antFactory
-Manager manager
-Executor executor
-PipeAdvertisement gateServicePipeAd
-PipeAdvertisement downloadServiceP
-ModuleSpecAdvertisement gateServic
-ModuleImplAdvertisement gateServic
-ModuleSpecAdvertisement download
+ AnthillPeerWS(Factory factory)
-PeerGroup startJxta()
-void setServices(PeerGroup grp)
-void joinGroup(AnthillGroup grp)
-PipeAdvertisement createGateServiceI
-PipeAdvertisement createDownloadSe
Nest nest
Thread
Executor
-Manager manager
-boolean executing
-AntView view
Executor()
+void setManager(Manager mr
void execute(AntView view)
+void run()
void waitStart()
void notifyStop()
anthill.nest.Manager
Manager
-Factory factory
-Queue queue
-int nthreads
-HashSet free
-HashSet busy
-String NTHREADS_PARAM="n
+ Manager(int nthreads)
+ Manager(Factory factory, St
+void add(anthill.AntView vie
+void execute()
void completed(Thread thread
+void clear()
anthill.AntView
AntView
-Nest nest
+ AntView(Nest nest, AntDescriptor aDesc,
+AntId createAntId()
+void addAnt(Ant ant)
+void move(NestId id)
+boolean storeReturnNest(AntId aid, NestId
+NestId getReturnNest(AntId aid, boolean re
+void addNeighbour(NestId id)
+void removeNeighbour(NestId id)
NestId nestId
NestId lastVisited
int TTL
Thread
GateServiceAdvPublisher
-DiscoveryService netDisco
-long expireTime
-Advertisement adv
+ GateServiceAdvPublisher(Di
+void run()

Figure 6.4–a jxtaimpl package UML Class Diagram

The *jxtaimpl* package is the JXTA based Anthill implementation presented along with this thesis. As stated before, just like any other possible Anthill implementation, *jxtaimpl* has been fulfilled implementing the set of interfaces described above.

In Figures 6.4−a,b,d, is depicted the UML class diagram describing the *jxtaimpl* package. Let us first consider Figure 6.4−a. As one can see, three sub packages are located within the jxtaimpl package, namely: *jxtaimpl.storage*, *jxtaimpl.ids* and *jxtaimpl.gui*.

- **jxtaimpl.storage**. This is the storage implementation used along with the JXTA based Anthill Project binding.
- **jxtaimpl.ids**. This package contains the classes implementing the identifiers used along with jxtaimpl.
- **jxtaimpl.gui**. This package is of minor interest at the moment, since it simply is a demo application written for testing purposes.

Let us now take into account the implementation classes wrapped in the *jxtaimpl* package.

- **jxtaimpl.AntView**. This class implements the formerly described *anthill.AntView* interface. What it does is mainly mapping all of its methods over the class implementing the *anthill.nest.Nest* interface.
- **jxtaimpl.Manager**. This is the *anthill.nest.Manager* implementation class. As stated above it is in charge of scheduling the received ants according to the desired policy. In the JXTA based Anthill implementation, this class is not concerned with ants execution: this is up to the class named *jxtaimpl.Executor*, whose behavior is treated in the *implementation* chapter.
- **jxtaimpl.AnthillPeerWS**. This class is in charge of boot strapping an Anthill Nest by launching the JXTA platform and starting the JXTA enabled services [PJJFPG] provided with this implementation. As explained in chapter 7, this is accomplished publishing JXTA advertisements for each provided service. This class is even in charge of instantiating all the other components of the project, according to the configuration parameters provided by the user via an XML configuration file (see chapter 7 for details).

- **jxtaimpl.PeerFinder**. This interface has to be implemented by those classes in charge of discovering remote nests via the PDP JXTA protocol [PJJFPG]. As one can notice in Figure 6.4−a, this interface is implemented by the jxtaimpl.AnthillPeerWS class.
- **jxtaimpl.GateServiceAdvPublisher**. Once the *AnthillPeerWS* instance has published the *GateService* advertisement, this will have to be periodically published back (see chapter 7 for details). The *jxtaimpl.GateServiceAdvPublisher* class is meant for performing this task.

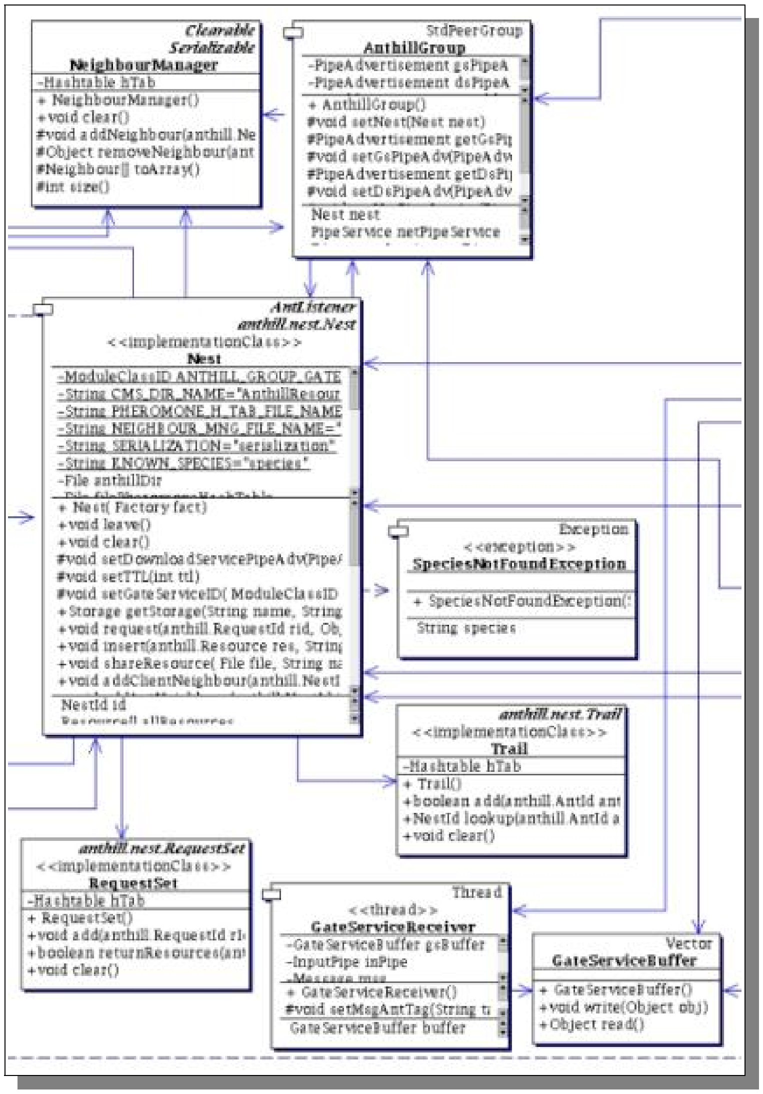

Clearable
Serializable
NeighbourManager
-Hashtable hTab
+ NeighbourManager()
+void clear()
#void addNeighbour(anthill.Ne
#Object removeNeighbour(ant
#Neighbour[] toArray()
#int size()
StdPeerGroup
AnthillGroup
-PipeAdvertisement gsPipeA
-PipeAdvertisement dsPipeA
+ AnthillGroup()
#void setNest(Nest nest)
#PipeAdvertisement getGsPi
#void setGsPipeAdv(PipeAdv
#PipeAdvertisement getDsPi
#void setDsPipeAdv(PipeAdv
Nest nest
PipeService netPipeService
AntListener
anthill.nest.Nest
<<implementationClass>>
Nest
-ModuleClassID ANTHILL_GROUP_GATE
-String CMS_DIR_NAME="AnthillResour
-String PHEROMONE_H_TAB_FILE_NAME
-String NEIGHBOUR_MNG_FILE_NAME="
-String SERIALIZATION="serialization"
-String KNOWN_SPECIES="species"
-File anthillDir
+ Nest(Factory fact)
+void leave()
+void clear()
#void setDownloadServicePipeAdv(Pipe
#void setTTL(int ttl)
#void setGateServiceID(ModuleClassID
+Storage getStorage(String name, String
+void request(anthill.RequestId rid, Ob
+void insert(anthill.Resource res, Strin
+void shareResource(File file, String na
+void addClientNeighbour(anthill.NestI
NestId id
Exception
<<exception>>
SpeciesNotFoundException
+ SpeciesNotFoundException(
String species
anthill.nest.Trail
<<implementationClass>>
Trail
-Hashtable hTab
+ Trail()
+boolean add(anthill.AntId ant
+NestId lookup(anthill.AntId a
+void clear()
anthill.nest.RequestSet
<<implementationClass>>
RequestSet
-Hashtable hTab
+ RequestSet()
+void add(anthill.RequestId ri
+boolean returnResources(an
+void clear()
Thread
<<thread>>
GateServiceReceiver
-GateServiceBuffer gsBuffer
-InputPipe inPipe
+ GateServiceReceiver()
#void setMsgAntTag(String t
GateServiceBuffer buffer
Vector
GateServiceBuffer
+ GateServiceBuffer()
+void write(Object obj)
+Object read()

Figure 6.4−b jxtaimpl package UML Class Diagram

Let us now consider the classes depicted in the second portion of the *jxtaimpl* package class diagram (figure 6.4−b). Firstly we observe three classes implementing three distinct interfaces located in the package *anthill.nest*, namely:

- **jxtaimpl.Nest**
- **jxtaimpl.RequestSet**
- **jxtaimpl.Trail**

Since these classes' semantic has already been explained, for details please refer back to section 6.4. The other classes depicted in figure 6.4−b only characterize the JXTA Anthill implementation and should not be thought as common to any other Anthill "binding". These are:

- **jxtaimpl.AnthillGroup***.* This class implements the concept of group. It is meant for providing references to the needed core−JXTA services, such as the *discovery service* and the *pipe service* [PJTO].
- **jxtaimpl.NeighbourManager***.* While a nest is alive it will periodically try to find out other peers via the *peer discovery protocol* (PDP) [PJJFPG]. All the gathered nest identifiers need to be somehow managed: the *NeighbourManager* class is in charge of performing this task.

The classes *jxtaimpl.GateServiceReceiver* and *jxtaimpl.GateServiceBuffer* are treated in the next section.

6.7 JXTAnthill: JXTA−Enabled Services

Each nest implementation has to provide at least one fundamental service to ants, the *gate* service. This service is described by the the *Gate* interface located in the *anthill.nest* package. Beyond this service, different Anthill implementations might need to extend the spectrum of *middle−level services* they provide: this is right what happened with the JXTAnthill implementation. JXTAnthill basically provides two distinct services. Both of them are classified as JXTA−enabled services [PJJFPG]. One is meant only for ants, while the other one is only available to the higher application built on top of JXTAnthill.

- **GateService***.* This is the JXTA−based *Gate Service* implementation, the only compulsory service for every Anthill implementation. Four different classes are involved in providing this JXTA−enabled service, namely:

  - *jxtaimpl.GateService*
  - *jxtaimpl.GateServiceReceiver*
  - *jxtaimpl.GateServiceBuffer*
  - *jxtaimpl.GateServiceServer*

The JXTAnthill *GateService* service has been shaped using a classical *producer−consumer* design pattern. The *GateService* class is essentially in charge of receiving foreign ants coming from the network layer. No processing has to be carried out at this level: once an ant wrapped into an *AntDescriptor* has been received it is passed to the class instance in charge of scheduling and execution issues (the *jxtaimpl.Manager*). So in JXTAnthill ants are received by the *GateServiceReceiver* instance, written into the *GateServiceBuffer* and read by the *GateServiceServer*.

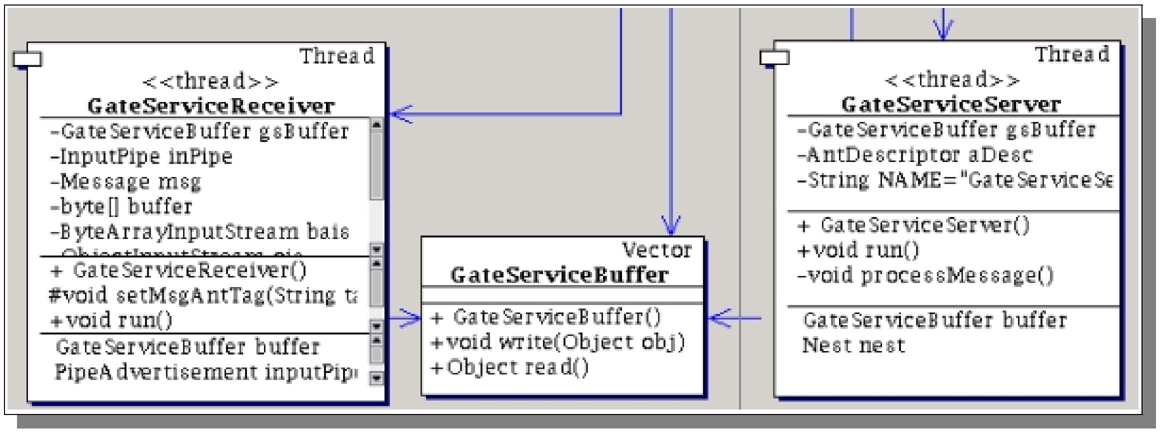


Figure 6.4−c GateService main components

- **DownloadService**. In the Anthill Project ants are high−level services built on top of the Anthill infrastructure. For demonstration purposes this thesis deals with an ant species designed for building a resource−sharing peer−to−peer system. In such a scenario it would not be advisable to let ants carry the resources they have found along with them through the network: think about what that would lead to if the resource was a very huge file. It would be better to have ants returning only resource descriptors, and letting the nest decide whether downloading that resource or not. In pursuit of this idea the *DownloadService* has been added to the set of JXTA−enabled services provided by each JXTAnthill Nest. The *DownloadService* is composed by three classes, namely: *jxtaimpl.DownloadService*, *jxtaimpl.DownloadManager*, *jxtaimpl.Downloader* (see figure 6.4−d). The *DownloadService* instance is in charge of instantiating and starting the *DownloadManager.* This component will wait for downloading requests originated by remote nests, so it can be thought as being the server side of the *DownloadService.* Its counterpart is the *Downloader*: should a nest decide to download a resource, it will leverage over its own *Downloader* instance in order to accomplish the task. The sequence of steps involved in an Anthill resource downloading is better described in the *implementation* section.

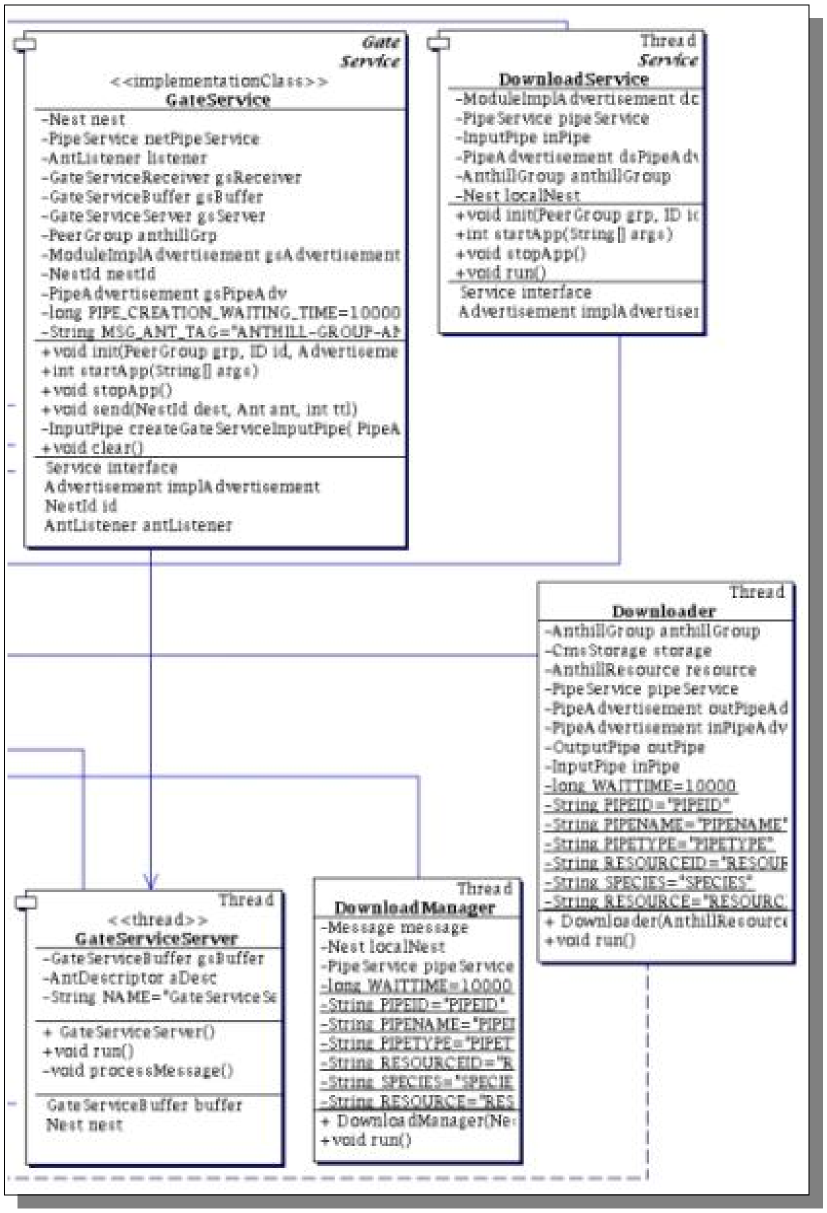

Gate Service
<<implementationClass>>
GateService
-Nest nest
-PipeService netPipeService
-AntListener listener
-GateServiceReceiver gsReceiver
-GateServiceBuffer gsBuffer
-GateServiceServer gsServer
-PeerGroup anthillGrp
-ModuleImplAdvertisement gsAdvertisement
-NestId nestId
-PipeAdvertisement gsPipeAdv
-long PIPE_CREATION_WAITING_TIME=10000
-String MSG_ANT_TAG="ANTHILL-GROUP-A
+void init(PeerGroup grp, ID id, Advertiseme
+int startApp(String[] args)
+void stopApp()
+void send(NestId dest, Ant ant, int ttl)
-InputPipe createGateServiceInputPipe(PipeA
+void clear()
Service interface
Advertisement implAdvertisement
NestId id
AntListener antListener
Thread Service
DownloadService
-ModuleImplAdvertisement dc
-PipeService pipeService
-InputPipe inPipe
-PipeAdvertisement dsPipeAdv
-AnthillGroup anthillGroup
-Nest localNest
+void init(PeerGroup grp, ID ic
+int startApp(String[] args)
+void stopApp()
+void run()
Service interface
Advertisement implAdvertise
Thread
Downloader
-AnthillGroup anthillGroup
-CmsStorage storage
-AnthillResource resource
-PipeService pipeService
-PipeAdvertisement outPipeAd
-PipeAdvertisement inPipeAdv
-OutputPipe outPipe
-InputPipe inPipe
-long WAITTIME=10000
-String PIPEID="PIPEID"
-String PIPENAME="PIPENAME"
-String PIPETYPE="PIPETYPE"
-String RESOURCEID="RESOUF
-String SPECIES="SPECIES"
-String RESOURCE="RESOURC
+ Downloader(AnthillResource
+void run()
Thread
<<thread>>
GateServiceServer
-GateServiceBuffer gsBuffer
-AntDescriptor aDesc
-String NAME="GateServiceSe
+ GateServiceServer()
+void run()
-void processMessage()
GateServiceBuffer buffer
Nest nest
Thread
DownloadManager
-Message message
-Nest localNest
-PipeService pipeService
-long WAITTIME=10000
-String PIPEID="PIPEID"
-String PIPENAME="PIPEI
-String PIPETYPE="PIPET
-String RESOURCEID="R
-String SPECIES="SPECIE
-String RESOURCE="RES
+ DownloadManager(Ne
+void run()

Figure 6.4–d jxtaimpl package UML Class Diagram

6.8 Package *jxtaimpl.storage*

Fundamental to the Anthill Project is the concept of *storage*. As previously noticed, entities ranging from pheromone to documents and URL references could all be arranged in a storage. Since the Anthill Project only defines a set of interfaces describing what a generic storage should provide (see package *anthill.storage* for details), developers are enabled to derive their own implementations starting from the given guide lines. For our purposes has been implemented a CMS−based storage [CMS]. CMS (Content Menagement Service) is a JXTA sub project for content handling and managing. Each content is associated with a JXTA content advertisement [PJJFPG] describing the resource itself. In order to retrieve shared documents, or to test the existence of a given resource, the documentary base can obviously be queried. The *jxtaimpl.storage* package consists of three classes, namely: the *CmsStorage*, the *AnthillResource* and the *Query* class.

- **jxtaimpl.storage.CmsStorage**. This is the class implementing the CMS−based resource storage. As one can see, it is an *anthill.storage.QueryStorage* implementation class. This class is essentially in charge of managing the set of resources a nest is willing to share. In order to limit the number of locally stored resources, a user−defined amount of disk space available to the storage can be assigned to each *CmsStorage* instance. Remember that each ant species is assigned to a distinct storage instance. As stated above, queries can be issued to a storage: a *CmsStorage* will accept only queries instantiating the *jxtaimpl.storage.Query* class.
- **jxtaimpl.storage.Query**. Since each resource is described by an advertisement (a *content advertisement*), the easiest way for querying the storage is by using couples shaped this way: *[Element−Name, Set of Values]*.
  - *Element−Name.* A *content advertisement* is an XML document containing information about the associated content shared within the peer group [PJJFPG]. A valid *element−name* is equal to one of the tags contained into the given *content advertisement*.
  - *Set of Values*. This second parameter is simply a sequence of values of interest for the given *element name*.

- **jxtaimpl.storage.AnthillResource**. Since ants are not allowed to move real resources from one nest to another, queries to a *CmsStorage* instance will only return resource descriptors. An *AnthillResource* plays this role: it merely describes what a resource is, highlighting details such as the content's length, the resource's logical description and its name. An *AnthillResource* contains as well instructions for remote nests to download the real resource via the *DownloadService*.

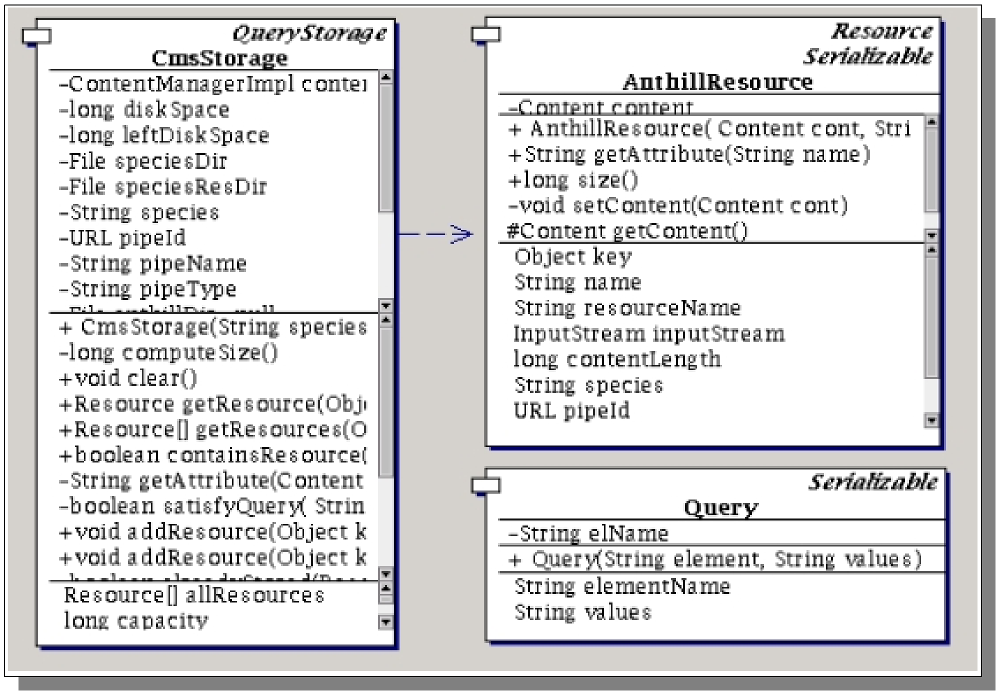


Figure 6.5 jxtaimpl.storage UML Class Diagram

6.9 Package *jxtaimpl.ids*

Each Anthill Project implementation has to provide its own identifiers implementation. Both the *AntId* and the *RequestId* extend the *GenericId* class reusing this way the same constructor. An *AntId* is originated starting from a valid *RequestId* instance, while a *RequestId* instance is obtained starting from a valid *NestId* instance. This way having an *AntId* or a *RequestId* it is possible to identify the querying nest. Since JXTAnthill Nests are JXTA peers, the most useful way for identifying a nest is using its *gate service pipe advertisement* as a unique identifier. This is an appropriate choice since *pipe advertisements* are already guaranteed to be unique in a JXTA network; furthermore, once such a nest identifier has been obtained, it can be immediately used for establishing a connection to the given nest without further processing.

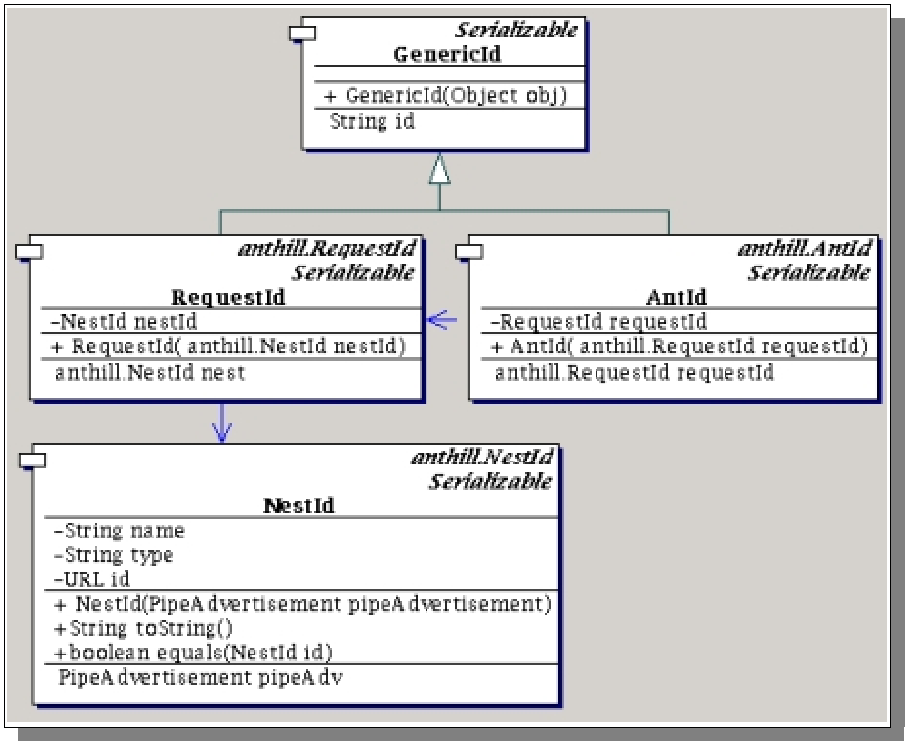


Figure 6.6 jxtaimpl.ids UML Class Diagram

6.10 Package *ants*

Besides the middle−level services exposed so far, the Anthill Project lets developers devise new higher level services based on the underlying platform. No restrictions are imposed about what these services should be meant for: the only necessary constrain is that all of them have to be shaped as *ants*. New services can be added to the Anthill framework simply implementing new ant species. Once an ant species has been devised, developers should place the associated code beneath the *ants* package. Ant species located in this position are guaranteed to be executed in a sandbox granting the minimum set of rights to the code. Theoretically what an ant should be able to do is strictly limited to interacting with the local *Nest* instance through the given *AntView,* nothing else: *ants* are treated just like Java™ Applets*.* A valid *ant* implementation should fulfill the following requirements:

- Must implement the *anthill.ant* interface.
- Ants have to be instantiated through a class implementing the *anthill.AntFactory* interface*.*

The *AntFactory.getAnts()* method will return an array of objects implementing the *Ant* interface. The number of the returned ants can change from implementation to implementation. As far as the *Ant* interface is concerned, the methods exposed in figure 6.8 have to be provided by every Ant implementation class.

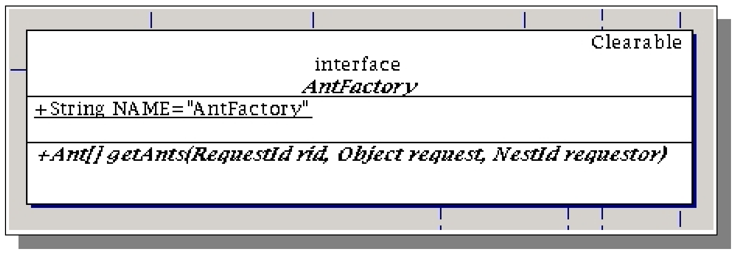

Clearable
interface
AntFactory
+String NAME="AntFactory"
+Ant[] getAnts(RequestId rid, Object request, NestId requestor)

Figure 6.7 anthill.AntFactory interface

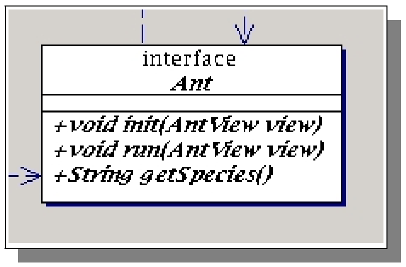


Figure 6.8 anthill.Ant interface

7 JXTAnthill: Implementation

7.1 Introduction

While the former chapter has dealt with the JXTAnthill Project design, mainly treating architectural issues, this one is concerned with some of the JXTAnthill Project most relevant and noticeable implementation details. In pursuit of this objective the explanation covers the main tasks performed by a generic JXTAnthill nest, from its start up process to its interaction with other JXTAnthill nests.

7.2 JXTAnthill: Nest Configuration

Each Anthill nest is extremely flexible due to the possibility of configuring it before starting the whole platform, by specifying a number of different parameters influencing the nest's behavior. Configuration is achieved through an XML configuration file. It is advisable to have an XML configuration file for each Anthill binding. It is not only needed for pointing out nest−specific configuration parameters, but it lets users decide which Anthill components should be plugged into the platform at run time. For example, as stated in the former chapter, many different implementations of one of the Anthill interfaces may be available to a nest, say the *anthill.nest.Manager* interface. This is the interface in charge of handling the *ant−scheduling* task. Since different implementations will provide different scheduling policies (round−robin, priority−based, etc.), one might choose to plug the most well−fitting *anthill.nest.Manager* implementation class into the Anthill Platform, according to his/her needs. Obviously, different flexibility levels may be experienced among different Anthill bindings: one could provide users with the ability to control a wide number of details while others might let users only define the minimum set of parameters needed. Figure 7.2 depicts a sample XML Anthill configuration file.

```
<?xml version="1.0" encoding="us-ascii"?>

<!ELEMENT SystemConfiguration (Component+, Parameter*)>
<!ATTLIST SystemConfiguration version CDATA #REQUIRED>

<!ELEMENT Component (Parameter*)>
<!ATTLIST Component name CDATA #REQUIRED>
<!ATTLIST Component class CDATA #REQUIRED>

<!ELEMENT Parameter EMPTY>
<!ATTLIST Parameter name CDATA #REQUIRED>
<!ATTLIST Parameter type CDATA #REQUIRED>
<!ATTLIST Parameter value CDATA #REQUIRED>
```

Figure 7.1 Factory DTD (factory.dtd)

```
<?xml version="1.0" encoding="us-ascii"?>
<!DOCTYPE SystemConfiguration SYSTEM "factory.dtd">

<SystemConfiguration version="1.0">

<!-- This section lets you specify which kind of Ant you want your
      Nest  to generate. -->
<Component name="AntFactory" class="jxtagnutant.GnutantFactory" />

<!-- This section is meant for setting which Manager instance should
      be in charge of scheduling ant axecution for your Nest. -->
<Component name="Manager" class="jxtaimpl.Manager">
      <Parameter name="nthreads" type="int" value="20"/>
</Component>

<!-- The NestTTL is your Ants' TTL value within the AnthillNetwork.
-->
<Parameter name="NestTTL" type="int" value="20"/>
```

```
<!-- A "yes" value means that you want your Nest to read information
     such Pheromone objects and Neighbour lists from configuration
     files if there are. Otherwise (a "no" value) brand new instances
     will be created for each session. -->

<Parameter name="serialization" type="String" value="yes"/>

<!-- This settings are about storage. The value is the space assigned
      to each storage instance expressed in Bytes. There has to be one
      entry for each ant species we are willing to accept in our Nest.
-->
<Parameter name="randomant" type="long" value="10000000"/>
<Parameter name="jxtagnutant" type="long" value="10000000"/>
<Parameter name="jxtant" type="long" value="10000000"/>


<!-- Here you can specify all the ant species your nest will accept,
      separated by a blank space -->
<Parameter name="species" type="String" value="jxtant jxtagnutant"/>

<!-- The Closeness value means how many answers we wish a Gnutant to
      return -->
<!-- The Capacity value means how many entries a GnutantPheromone will
      be able to store. A zero value means infinite -->
<Parameter name="GnutantPheromoneCloseness" type="int" value="20"/>
<Parameter name="GnutantPheromoneCapacity" type="int" value="0"/>

<!-- This is the time in milliseconds elapsing between two following
      discovery requests.  -->
<Parameter name="sleeptime" type="int" value="15000"/>

<!-- The following parameter specifies where the configuration files
     created by "jxtanthill" should be placed. You have to point out
     a path without the final path-separator char. -->
<Parameter name="confdir" type="String" value="/root"/>

<!-- Gate Service Adv life time in milliseconds. -->
<Parameter name="GateAdvLifeTime" type="int" value="1800000"/>


</SystemConfiguration>
```

Figure 7.2 JXTAnthill XML configuration file

Figure 7.1 shows the DTD that the Anthill XML configuration file should reflect. There are three elements:

- **System Configuration***.* A system configuration consists of a not empty set of *Components* along with a potentially empty set of *Parameters*.
- **Component***.* Each *Component* element is characterized by a couple of attributes, namely:
  - **Name***.* This attribute is needed for identifying the interface this *component* element refers to. It is not required to highlight the interface's fully qualified name. The *name*'s value is merely used for indexing purposes.
  - **Class***.* The *class* attribute tells the platform which class is in charge of actually implementing the above specified interface. In order to retrieve the *class* attribute's value, the application code should be aware of the *name* attribute's value used as index.

Since components are used in order to define which class implements a given interface, and since classes might need to be initialized using a given set of parameters, a *component* element accepts a potentially empty set of *parameter* elements as well.

- **Parameter***.* The *parameter* element is used for defining elementary−type configuration parameters. The following attributes characterize a *parameter* element:
  - **Name***.* This is the logical name used for indexing the parameter itself. It has to be necessarily unique in the given configuration file.
  - **Type***.* This attribute defines the *parameter*'s type (int, string, long, etc.).
  - **Value***.* This is simply the value associated with the *parameter* itself.

AntFactory Component. The first component itemized in figure 7.2 is the one named *AntFactory*. This component tells the platform which factory class has to be used for instantiating ants, or in other words this component defines which is the high−level service provided by the nest.

Manager Component. This component defines which class implementing the *anthill.nest.Manager* interface should be used. This component needs one input parameter: the maximum number of admitted threads. What this exactly means is explained in section 7.4.

Nthreads Parameter. This parameter specifies how many threads the Manager is allowed to instantiate for executing ants.

Nest TTL Parameter. The JXTAnthill implementation lets users customize the Time−to−Live value assigned to ants. It has been decided to give users control above this parameter since JXTAnthill may be deployed in different scenarios characterized by deeply different kinds of topologies and dynamics: having an "hardwired" TTL value would have led to a too much inflexible result. Another approach would have been letting the platform itself be in charge of tuning the TTL value as well as possible, but probably in this case nests should have been provided with some kind of knowledge about topological issues or the kind of services ants were meant to perform. Having the TTL parameter configurable by users seems to be a good trade off between simplicity and flexibility. The TTL is expressed as number of hops in the JXTAnthill network.

Serialization Parameter. While a JXTAnthill nest is alive, it interacts with its hosting environment collecting information about the nests it is surrounded by. Furthermore nests are in charge of managing data structures on behalf of ants: the pheromone objects. All this information is not required to be handled in any specific way, so different Anthill implementations are expected to treat discovered information differently: they could decide to permanently store these data structures for reusing them in the future, or not. It is important to highlight there is not an a priori most fitting approach to resource management, since JXTAnthill can be used for building a pure peer−to−peer system both in a highly dynamic environment and in a static one. Let us suppose to deploy an Anthill implementation in an *ad−hoc* and highly dynamic network. In such a scenario it would not be of any interest saving information describing each session's neighbourhood: all likely at the next start up the nest will have a completely different scope. Differently, in an environment were nodes are not supposed to be dynamic, periodically storing information about the nest's neighbourhood becomes useful. Suppose a nest crashes: at the next start up it would be bereaved of discovering all its neighbours once again thanks to the previously collected and stored information. Due to this set of reasons the JXTAnthill implementation grants users the ability to control this behavior by setting the *serialization* parameter. A *"yes"* value means information gets locally stored, a "*no*" value means the opposite.

Storage Settings. These *parameters* allow users to specify how much disk space is available to each ant−species storage instance. There has to be one entry per ant species.

Species Parameter. The *species* parameter simply states which ant species the nest is willing to accept.

Closeness & Capacity Parameters. These are parameters specific to the ant species used along with the JXTAnthill implementation. They define properties concerning the pheromone data structure used by ants.

Sleeptime Parameter. This parameter defines how much time elapses between two consecutive discovery sessions. See section 7.3 for more details.

Confdir Parameter. This parameter's value identifies where configuration files should be locally placed.

GateAdvLifeTime. This parameter states how long it will take before one nest's Gate Service advertisement expires, both in the local and remote caches.

All these parameters and components are read from the configuration file and later on obtained through an *anthill.factory.Factory* instance.

7.3 JXTAnthill: Nest start up

The class responsible of a nest's booting process is the *jxtaimpl.AnthillPeerWS* class. When instantiated it is provided with an *anthill.factory.Factory* instance it will use to obtain all the needed configuration parameters. Besides properly setting all the parameters itemized in the XML configuration file, this class is in charge of instantiating all the other modules needed by the JXTAnthill platform. A detailed representation of all the performed steps is depicted in the UML Nest Start Up Sequence Diagram (Appendix A). What follows is a brief and literal description of the process:

1. Information such as the actual *Manager* implementation class to be used, the *AntFactory* and the TTL value are obtained from the given *Factory* instance.
2. Some of the needed classes are instantiated (*NeighbourManager, Nest).*
3. All the advertisements needed for instantiating the JXTA−enabled services provided by the nest are created [PJJFPG]. Advertisements, as stated in the chapter dealing with the JXTA Technology, are XML documents used for publishing the existence of heterogeneous resources. There are advertisements publishing peers, peer groups, pipes, rendezvous nodes etc.

4. The *AnthillGroup* advertisement is created. This is the JXTA peer group advertisement needed for initializing and booting the nest itself: this XML document itemizes all the services the nest has to provide as member of the group.
5. The *AnthillGroup* is initialized using the group advertisement formerly created. At this step the nest can be already thought as being part of the JXTAnthill network.
6. The AnthillGroup advertisement and the GateService advertisement are published. Publishing is necessary for spreading in the JXTAnthill network the knowledge of a new nest birth.
7. A *jxtaimpl.GateServiceAdvPublisher* is instantiated. This class is in charge of periodically publishing the GateService advertisement.

Once these steps have been successfully accomplished, the *AnthillPeerWS* instance starts engaging a fundamental task: the discovery of remote *nests.* This class implements the *jxtaimpl.PeerFinder* interface and extends the Java™ *Thread* class. In order to discover other nests, JXTAnthill relies on one of the core JXTA protocols: the *Peer Discovery Protocol* (PDP) [PJJFPG]. According to the *sleeptime* parameter described in the former section, this instance will periodically issue a PDP query message. What this query message looks for, are JXTA *pipe advertisements* whose name has to be equal to the one used for identifying the *Anthill GateService pipe advertisement*. Each discovered pipe advertisement is translated into a valid JXTAnthill Nest identifier and saved into the NeighbourManager instance.

7.4 NeighbourManager Implementation

The *jxtaimpl.NeighbourManager* class hides an hash table in which all the collected nest identifiers are arranged. In this data structure no duplicates are admitted: every time the *addNeighbour(anthill.NestId id)* method is called, the *NeighbourManger* instance will test whether the given identifier is already stored in the local hash table or not: the identifier gets locally stored if and only if it is not already in. Neighbors identifiers can be removed from the *NeighbourManager*'s internal data structure as well. This happens when a nest tries to get connected with a peer that turns out to be unreachable. If so its identifier gets flushed out from the local cache. If that nest came back to life in the JXTAnthill network, it would publish back its own *GateService* advertisement becoming “visible” again to the other nests.

7.5 GateServiceAdvPublisher Implementation

JXTA advertisements are spread through the network using the Peer Discovery Protocol, the means by which JXTA resources can be published and discovered by JXTA−based peers [PJJFPG]. Since there are no assumptions concerning the dynamics each peer will expose, to prevent advertisements from lingering in the network even after their publishing peer has crashed down or has simply been disconnected from the network, to each advertisement is assigned a customizable life time. The JXTAnthill user has control over this value since it can be configured through the XML configuration file formerly discussed. The ability of tuning the GateService advertisement's lifetime is fundamental, as it prevents nests from maintaining in their neighbour tables no more useful information. The task of periodically publishing back the *GateService* service advertisement is accomplished by a *jxtaimpl.GateServiceAdvPublisher* instance.

This is merely a thread which will periodically republish the *GateService* pipe advertisement in the JXTAnthill network. The period is given by the *GateAdvLifeTime* parameter value, and the *jxtaimpl.GateServiceAdvPublisher* object is instantiated by the *jxtaimpl.AnthillPeerWS* instance.

7.6 Nest Implementation

Once the JXTA platform has been started, and both the *GateService* and the *AnthillGroup* advertisements have been successfully published, the attention gets focused on the *jxtaimpl.Nest* instance. This class implements two interfaces:

- *anthill.nest.Nest.*
- *anthill.nest.AntListener*.

The *jxtaimpl.Nest* constructor accepts an *anthill.factory.Factory* instance as input parameter. Once the constructor is invoked it will check whether all the necessary configuration files can be locally found or not, starting from the current directory. This check occurs if and only if the *serialization* parameter has a "yes" value. During each session, the following information is saved:

- The *pheromone* objects.
- The set of *neighbours*.
- The *storage* instances.

Figure 7.3 depicts how these data structures are locally arranged.

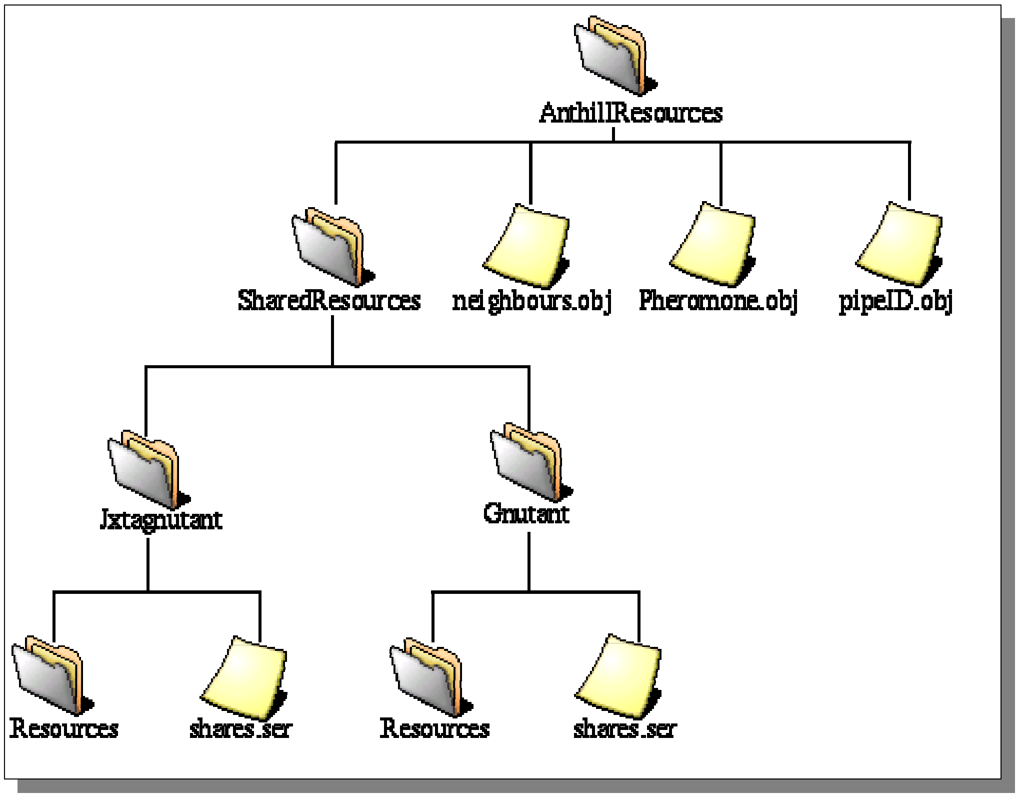


Figure 7.3 Configuration Directory structure

The *nest* instance is only in charge of managing the top level resources, namely the files *neighbours.obj*, *pheromone.obj* and *pipeID.obj*. The *SharedResources* folder contains each ant species' storage. In order to keep the local information as much updated as possible, upon changes in the pheromone data structures or in the set of known neighbours, the *nest* instance saves these data structures to disk. This way if the nest crashed down, it would be restarted with the most recent knowledge available about the network. As stated before the *jxtaimpl.Nest* class implements the *anthill.nest.AntListener* interface. This implies that all the received ants will be delivered to the *nest* instance *by* invoking its *deliver(AntDescriptor aDesc)* method. The received ants are passed to the *jxtaimpl.Manager* instance which is described in the reminder of this chapter.

7.7 GateService Implementation

The means by which virtual ants can move from one nest to another in the JXTAnthill network is given by the *GateService*. The JXTAnthill implementation introduces a JXTA−based implementation of this service. What this means is that JXTA pipes are used as the standard communication primitive among nests. JXTA pipes have been preferred to any other communication mechanism due to their interesting set of properties. Although JXTA pipes are classified as unreliable, they still offer many other built in facilities fundamental to *peer−to−peer* systems development. Using JXTA pipes it is no more necessary to mind about firewall and NAT related issues: nests are enabled to communicate with one another always using the most suitalble protocol available (TCP/IP, HTTP over TCP/IP if traversing a firewall or a NAT, etc.). Even further, using JXTA pipes it is not required to face routing related problems since messages are exchanged using the *peer endpoint protocol* (PEP) formerly described in the chapter about the JXTA Technology [PJVN].

The *jxtaimpl.GateService* class implements both the *net.jxta.service.Service* and the *anthill.nest.Gate* interfaces. This class is instantiated invoking the *init* method over the *jxtaimpl.AnthillGroup* instance. The *jxtaimpl.GateService* class is one of the three implementation classes needed to provide the *GateService* service. As seen in the previous chapter the other ones are:

- *jxtaimpl.GateServiceReceiver*
- *jxtaimpl.GateServiceBuffer*
- *jxtaimpl.GateServiceServer*

All these classes are instantiated during the *GateService* initialization process. The most noticeable method belonging to this class is the *public void send(NestId destId, Ant ant, int ttl)* method. This has to be invoked in order to send ants from the hosting nest to the one identified by the given NestId. What this method does is try and get connected to the destination nest using the given NestId as a valid JXTA pipe advertisement. The ant will be sent if and only if the ttl *(Time−to−Live)* value is greater than zero. The message is composed by two elements:

- The ANT_MESSAGE_TAG.
- The serialized *AntDescriptor* wrapping the virtual ant.

The ANT_MESSAGE_TAG is simply a string identifying the message's content type: an ant. The *AntDescriptor* is a wrapper used for exchanging ants between nests. Before being sent it is converted into a raw stream of byte, and then attached to the ANT_MESSAGE_TAG. The *jxtaimpl.GateService* instance will try and send the message for a given amount of time. If something goes wrong an *IOException* is thrown.

At the moment the JXTAnthill *GateService* implementation only uses *point−to−point JXTA pipes* for exchanging messages: it could be interesting to test which kind of behavior would emerge using *propagate JXTA pipes* implementing a one−to−many communication pattern among nests. If the *jxtaimpl.GateService* class is in charge of sending ants, the receiving task is up to the *jxtaimpl.GateServiceReceiver* class. This class is in charge of writing each received ant into a synchronized buffer of messages, shared with the *jxtaimpl.GateServiceServer* instance. It is this object that will extract ants from messages and will deliver them to the class instance implementing the *anthill.nest.AntListener* interface. The whole process is graphically described in the GateService UML sequence diagram presented in Appendix A.

7.8 Manager & Executor Implementation

Even if upon receipt ants are passed to the *jxtaimpl.Nest* instance, this class does not deal with ants execution. For this purpose the *anthill.nest.Manager* interface is implemented by the *jxtanthill.Manager* class. This class is instantiated using as input parameter the *nthreads* configuration parameter examined in section 7.2. As stated before this parameter defines the maximum number of threads this class is allowed to instantiate for executing ants. Its thread pool is represented by a pool of *jxtaimpl.Executor* instances. When ants are received they are associated with an *Executor* instance, if available, otherwise the ant is put into a waiting queue. Ants are actually executed by an *Executor* instance on behalf of the *Manager*.

7.9 DownloadService Implementation

In the JXTAnthill implementation ants are not allowed to move resources through the Anthill network: they can only carry along resource descriptors. These descriptors can be later on used for actually downloading the discovered resource by the JXTA−enabled *DownloadService* service. The UML sequence diagram describing an hypothetical download session, can be found in Appendix A. In order to download a resource, the *Nest.downloadResource(...)* method should be invoked. This causes the local nest to instantiate a *jxtaimpl.Downloader* object passing it the resource descriptor identifying what should be downloaded. The resource descriptor wraps a valid JXTA pipe advertisement. This is the pipe advertisement the nest owning the resource has used for opening its *DownloadService input pipe*. The *jxtaimpl.Downloader* instance uses this pipe advertisement for opening its *DownloadService output pipe*. What follows describes what happens when instantiating this class:

```
outPipeAdv = (PipeAdvertisement)AdvertisementFactory.newAdvertisement(
                   PipeAdvertisement.getAdvertisementType() );
outPipeAdv.setName( resource.getPipeName() );
URL url = resource.getPipeId();
outPipeAdv.setPipeID( (PipeID)IDFactory.fromURL(url) );
outPipeAdv.setType( resource.getPipeType() );

inPipeAdv = (PipeAdvertisement)AdvertisementFactory.newAdvertisement(
                   PipeAdvertisement.getAdvertisementType() );
inPipeAdv.setName( "tempPipe" );
inPipeAdv.setPipeID(IDFactory.newPipeID(anthillGroup.getPeerGroupID()));
inPipeAdv.setType( PipeService.UnicastType );
```

Figure 7.4 DownloadService initialization

Firstly it creates an output pipe using the *DownloadService pipe advertisement* the resource descriptor encloses. This pipe is used to tell the remote *DownloadManager* both which resource is required and which pipe advertisement it should use for uploading the resource: remember we are dealing with uni−directional pipes. As a second step, a temporary pipe is instantiated: this is the pipe used for actually reading in the resource as a byte stream. Once the connection has been successfully established the *Downloader* will send its counterpart a message enclosing the following tags:

- *PipeID*
- *PipeName*
- *PipeType*
- *ResourceID*
- *AntSpecies*

The first three elements describe the pipe advertisement that the *DownloadManager* instance should use for uploading the actual content associated with the resource identified by the fourth message element. The fifth message tag simply states which ant species the requested resource belongs to. Right after the message has been sent the *jxtaimpl.Downloader* instance keeps on passively waiting for the *jxtaimpl.DownloadManager* to upload the resource's content. Once the content has been completely read in from the input pipe, the *Downloader* will update the local storage with the acquired resource.

7.10 AnthillResource Implementation

As one could have argued, resource descriptors actually play a fundamental role in the process of downloading a real resource in the JXTAnthill implementation. The *jxtanthill.storage.AnthillResource* class is the descriptor used in our Anthill binding. It contains information about the actual resource, such as a logical description, its logical name, its unique identifier, the content's length and the pipe advertisement each remote nest should use for downloading the content. These advertisements are not published by every JXTAnthill nest, but simply wrapped into resource descriptors, thus bereaving every other nest of the burden to manage and cache a huge amount of potentially not useful data. In this way nests will handle the GateService service pipe advertisement only for downloading a resource.

7.11 Identifiers Implementation

The *jxtaimpl.ids* package contains all the identifier implementations needed in the JXTAnthill Anthill binding.

- **NestId**. As explained in the previous chapter each Anthill nest is identified by its own *GateService* pipe advertisement. So a valid *jxtaimpl.ids.NestId* will consist of:

  - The *pipe identifier*.
  - The *pipe's logical name*.
  - The *pipe type*.

  The last two components are represented as strings, while the first one is internally stored as a valid URL. By invoking the *NestId.getPipeAdv()* a JXTA pipe advertisement is automatically returned. Two *jxtaimpl.ids.NestId* instances equal if and only if their URLs equal with each other.

- **RequestId***.* A request identifier is initialized using the requesting nest's identifier. This way each request can be mapped to its originating nest. This class's constructor relies on the *jxtaimpl.ids.GenericId* class constructor.
- **AntId***.* An ant identifier requires as input parameter the identifier of the request it has to satisfy. So, given an ant identifier the ant's nest can be discovered.
- **GenericId***.* This class provides the constructor both the *RequestId* and the *AntId* classes use. This method needs an *Object* as input parameter, and simply generates an identifier shaped as a string consisting of two main parts:
  - The provided object's string representation.
  - A randomly generated value.

These identifiers are guaranteed to be unique in the JXTAnthill network. A JXTAnthill nest is identified by the JXTA pipe identifier associated with the pipe it uses for providing the *GateService* service. This JXTA identifier is already guaranteed to be unique, so will be the nest identifier as well.

A request identifier is derived starting from the originating nest identifier and a randomly generated value, so it is guaranteed to be unique until the pseudo−random number generator engine will wrap around starting reproducing the same sequence of values. The same considerations can be applied to ant identifiers: they are obtained starting from a request identifier and a randomly generated value.

7.12 Storage Implementation

In order to permanently store different kinds of information locally to a nest, the Anthill Project uses the concept of *storage*. In the Anthill Project each storage implementation has to be compliant with the interfaces provided by the *anthill.storage* package. In order to manage documents, the JXTAnthill binding defines a CMS−based storage implementation [CMS]. The *jxtaimpl.nest* class is in charge of instantiating a storage when an ant requires it. The following set of parameters is passed to the *jxtaimpl.storage.CmsStorage* constructor:

- The ant species the requiring ant belongs to.
- The amount of disk space granted to this storage instance. This parameter is specified in the XML configuration file depicted in figure 7.2.
- A valid JXTA pipe identifier.
- A pipe name.
- A valid JXTA pipe type.

The last three parameters need an explanation. As already stated, the JXTAnthill binding does not allow ants to collect real resources while travelling through the network: ants can only gather resource descriptors. Once the ant is returned to its home nest, the resource might be downloaded through the JXTA−enabled *DownloadService* service. In order to do this every *CmsStorage* has to be associated with a valid JXTA pipe advertisement that remote nests will use to download resources. This advertisement is added to every resource descriptor (*jxtaimpl.storage.AnthillResource*) the *CmsStorage* instance will return to querying ants. Once a *CmsStorage* is instantiated, if necessary it creates its own folders for storing resources beneath the *SharedResources* folder (see Figure 7.3). If there are previously saved resources, it computes the amount of already used disk space. Every time a resource has to written to disk, the *CmsStorage* will firstly check whether there is enough disk space left or not, potentially throwing a *CapacityException*: the resource is permanently added if and only if the test will give a positive result. The *CmsStorage* class only accepts query objects belonging to the *jxtaimpl.storage.Query* class. This class' instances consist of an element name and a set of keywords. Locally to a nest every resource is described by an XML document, a *ContentAdvertisement*: upon each query receipt the storage will look for resources whose *ContentAdvertisement* exposes at least one of the specified keywords for the given element name. If something useful is found, an *AnthillResource* instance is generated and returned to the querying ant.

7.13 Rendezvous and Relay Nests

As stated in chapter 4, JXTA peers need a way for dynamically discovering one another. This task is accomplished through the synergic usage of both the *peer discovery protocol* (PDP) and a particular kind of peer: the *rendezvous peer*. Rendezvous peers are fundamental to the peer discovery protocol since they essentially act as a repository of advertisements. Each JXTA peer has to be booted with at least one rendezvous peer, or it may be enabled to dynamically discover one. Every time a peer publishes an advertisement through the PDP, that advertisement will be cached by the rendezvous peer(s) the publishing node is connected to. As already explained, and as figure 7.5 depicts, JXTA rendezvous peers even play an important role in the process of requests propagation among peers: they are the only peers enabled to widen a request's scope by forwarding it to both other rendezvous and simple JXTA peers.

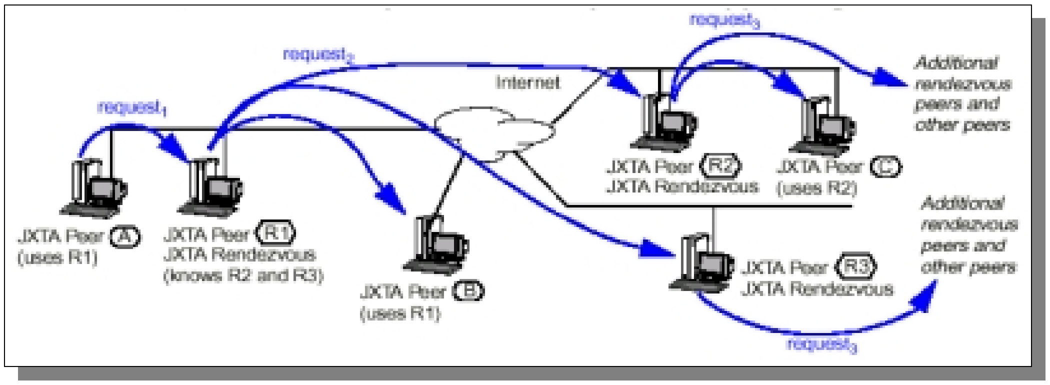

request$_3$
request$_2$
request$_1$
Internet
Additional rendezvous peers and other peers
JXTA Peer (R2) JXTA Rendezvous
JXTA Peer (C) (uses R2)
JXTA Peer (A) (uses R1)
JXTA Peer (R1) JXTA Rendezvous (knows R2 and R3)
JXTA Peer (B) (uses R1)
JXTA Peer (R3) JXTA Rendezvous
Additional rendezvous peers and other peers
request$_3$

Figure 7.5 JXTA Rendezvous peers role

Since JXTAnthill nests are JXTA peers, they need to rely on at least one rendezvous nest. In order to have a nest acting as a rendezvous peer it is sufficient to configure it as a rendezvous through the JXTA Configuration Panel displayed when booting the peer. Even rendezvous nests should be configured to use other nests as their own rendezvous nests. At the moment JXTAnthill nests are not capable of dynamically binding to newly discovered rendezvous nests, so they may only accept a static list of nests provided through the JXTA Configuration Panel. Now we can envision an initial topology the JXTAnthill network should reflect, in which rendezvous nests are as much as possible aware of one another. In this way rendezvous nests will give life to an highly connected network that will maximize each nest's discovery capability. Rendezvous nests can be thought as the cores of clusters of nests: *a cluster of nests is made up by nests configured with the same set of available rendezvous nodes*. This way the initial topology of the JXTAnthill network would be like the one depicted in figure 7.6.

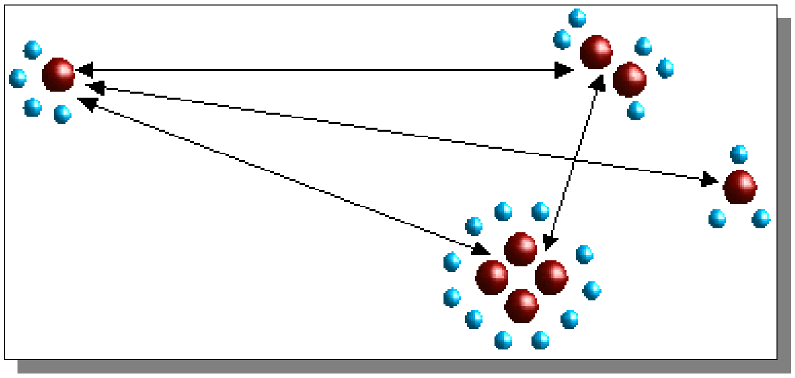


Figure 7.6 JXTAnthill network initial topology. Rendezvous nests are depicted bigger than simple nests. The bidirectional arrows represents *highways*: two clusters are connected by an arrow if they are aware of each other.

In the above representation rendezvous nests are depicted bigger than simple nests. As one can see they are surrounded by clouds of simple nests and interconnected with one another by *highways*. The existence of an highway between two rendezvous nodes does not mean there is an already established connection between them: what it simply means is that the two peers are aware of each other. It is interesting to highlight that this is only the JXTAnthill network initial configuration: as nests discover one another the complexity of the graph grows dramatically, and this has a great impact on the scope of requests. It is important to point out this is only one of the many initial configurations we can envision for the JXTAnthill network, absolutely not the only one.

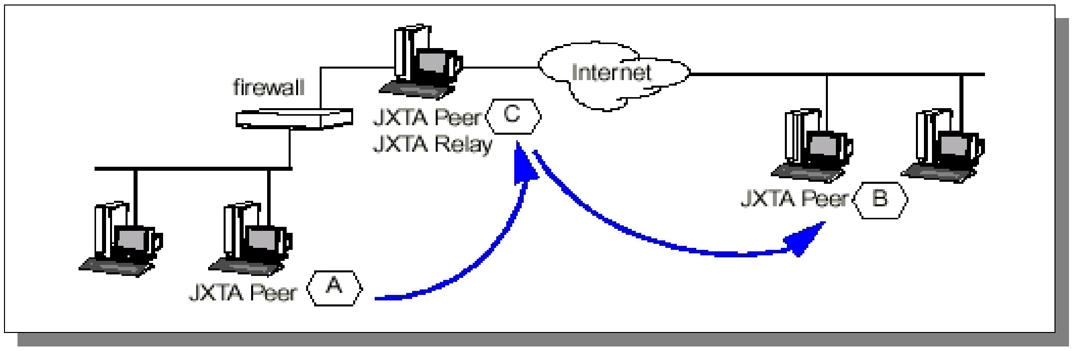
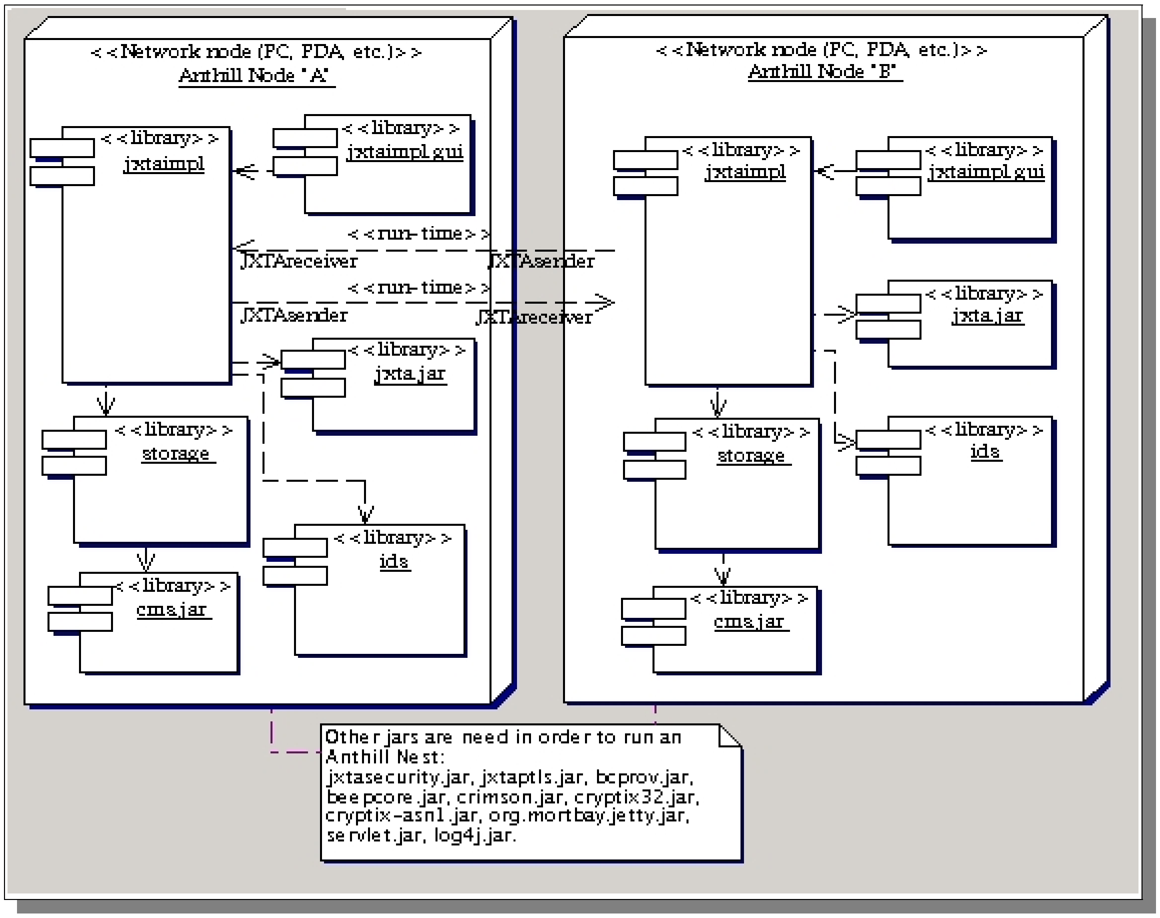


Figure 7.7 Relay nests' behavior

Since JXTAnthill nests might be physically located on different networks potentially divided by firewalls, it is fundamental to configure nests with a set of *relay nests*. As explained in chapter 4, these are peers in charge of relaying messages from one peer to another merely acting as traditional gateways. They even collect and cache routing information: this way, if a peer was unable to directly route a message to its destination, it would leverage on its relay peer(s). A JXTAnthill nest can be instructed to act as a relay peer via the JXTA Configuration Panel.

7.14 JXTAnthill Deployment



Other jars are need in order to run an Anthill Nest:
jxtasecurity.jar, jxtaptls.jar, bcprov.jar, beepcore.jar, crimson.jar, cryptix32.jar, cryptix-asn1.jar, org.mortbay.jetty.jar, servlet.jar, log4j.jar.

Figure 7.8 JXTAnthill UML Deployment Diagram

Being each JXTAnthill nest a pure JXTA peer, it can be deployed on any device provided with all the necessary libraries and having a digital heart beat. Obviously we firstly need the Anthill jar file containing the JXTAnthill implementation as well as. The bulk of libraries required by the JXTA platform. The most relevant jars are itemized in figure 7.8.

These libraries are mainly needed by the JXTA platform and so they might change in time according to JXTA developers' needs. For this reason the provided list has not to be thought as definitive at all. It does not matter which kind of connectivity is available to the device hosting the JXTAnthill nest since JXTA is completely independent from issues like this. The only imposed requirement is about having a Java™ Virtual Machine available on the device.

7.15 Conclusions & Future Improvements

The Anthill Project aims to define a framework to support the *design*, *implementation*, and *evaluation (testing)* of P2P applications. The Anthill Project is rooted in concepts borrowed from *complex adaptive systems* (CAS), since we think peer−to−peer systems can be thought as instances of CAS: they expose properties typical to complex systems, such as the absence of whatsoever form of centralization, a strong interaction among their basic components (the network nodes), and large scale and extreme dynamism of their operating environment [BMM−09−01]. The Anthill framework supports P2P applications based on the *multi−agent system* paradigm (MAS). What this means is that services have to be implemented developing *agents* capable of moving through the Anthill network. These mobile agents are called *ants*. Anthill uses a terminology derived from the ant colony metaphor. An Anthill network is made up of *nests*, devices running the Anthill runtime environment and provided with some kind of connectivity. Each nest is in charge of handling requests issued by higher level applications. All these requests entail the creation of a new ant belonging to a species able to satisfy the request itself. Once created, ants are in charge of executing their own task observing their local environment, interacting with it and performing simple local computations. In Anthill emergent behavior manifests itself as *swarm intelligence* whereby the collection of simple ants of limited individual capabilities achieves “intelligent” collective behavior [BMM−09−01]. Both the run−time environment and the simulation environment expose the same API, so testing a newly developed ant species does not entail writing the same code twice.

This thesis deals with the first implementation of the Anthill Project’s run−time environment. This implementation is written in Java™ and based on a new technology: JXTA. JXTA is an open−source project started in May 2001 and promoted by Sun Microsystems, Inc. Project JXTA defines a new platform for developing peer−to−peer applications, and guarantees the following properties [PJTO]:

- *Interoperability.* JXTA technology is designed to enable interconnected peers to offer services to each other seamlessly across different P2P systems and different communities .
- *Platform Independence.* JXTA technology is designed to be independent of programming languages, system platforms, and networking platforms.
- *Ubiquity.* Each JXTA peer can be deployed on any device having a digital heart beat.

We chose the JXTA Technology as the starting point for our Anthill run−time environment implementation, due to the enormous set of built−in facilities it provides: peer and peer group establishing and management, *ad−hoc* message routing, firewall and NAT traversing, virtual communication channels etc. The proposed JXTA−based Anthill implementation is called JXTAnthill. The implemented run−time environment has been tested with the Gnutant ant species, an ant species implementing a file−sharing service (chapter 8). At the moment JXTAnthill provides all the services ants should need for accomplishing their own tasks, but improvements are still expected, especially for gaining in performance. Here are itemized the topics we feel as being of major relevance.

Dynamic Discovery & Binding to new Rendezvous/Relay nests. At the moment, once an Anthill nest is started, it has to be configured with a static set of rendezvous and relay peers. Rendezvous peers are useful for augmenting each nest's degree of knowledge about the environment it is in. The more rendezvous nodes are available to a peer, the larger its own neighbourhood will become. This is because rendezvous nodes play a fundamental role to the JXTA discovery process used by JXTAnthill for finding other nests in the network. The same set of considerations can be applied to relay nodes, peers used for routing purposes. So, future JXTAnthill implementations should work over this topic, enabling JXTAnthill nests to dynamically bind to newly discovered rendezvous and relay nests.

Dynamic Downloading of Unknown Ant Species. We expect that future JXTAnthill implementations will be able to dynamically download the code of ants not yet locally installed. This way while a nest is alive, new ant species will be transparently added to the set of already known ants thus extending the set of high−level services available to the nest.

8 Example Application and Simulation Results

8.1 Intoduction: Technical Report Sharing

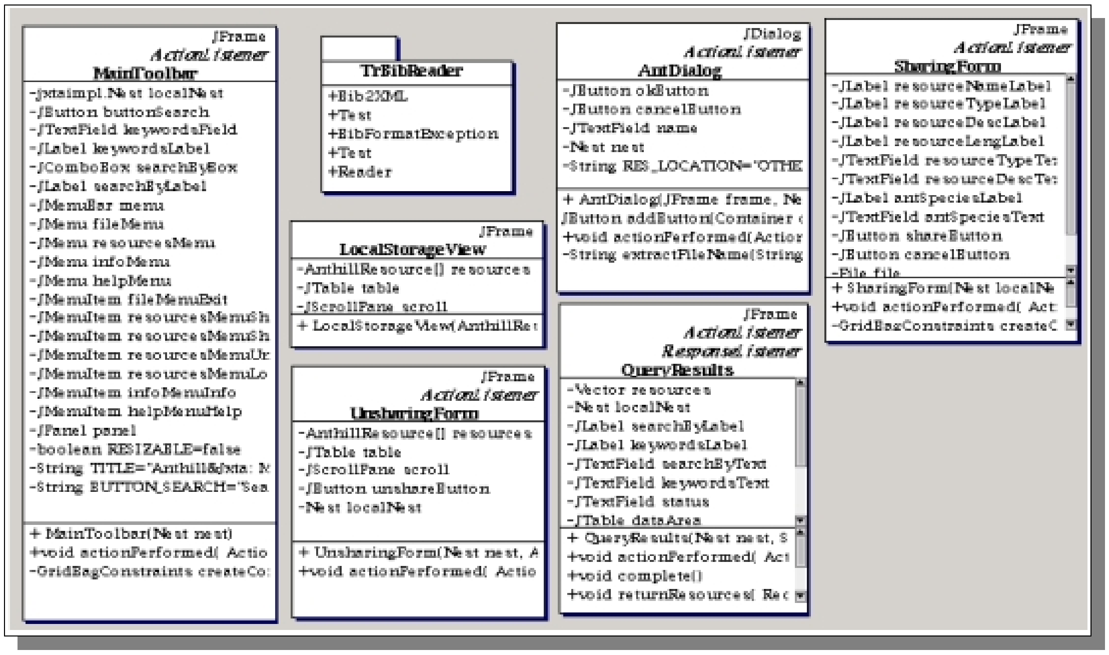


Figure 8.1 techrep package UML Class Diagram

The Anthill Project JXTA binding defines and implements a middle level infrastructure meant for granting virtual ants a basic set of services. In this architecture the only compulsory service is the *GateService* service, required by ants for moving from one nest to another in the Anthill network; beyond this service, different bindings of the Anthill project may provide other specific services not useful to every ant species. Stated this, in the Anthill Project's architecture, ants correspond to high level services: we can foresee different ant species implementing a classical document sharing service but with inherently different logic, ant species implementing a load balancing service, etc. On top of these two layers developers can place their front−end applications, the interfaces standing between users and the Anthill nest.

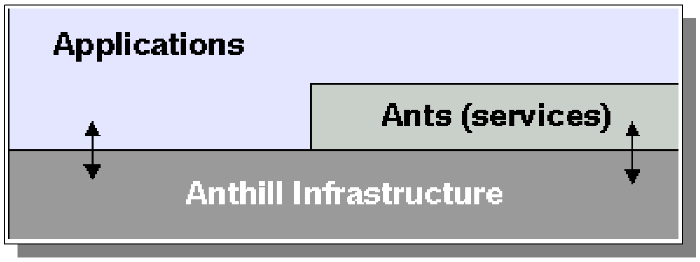


Figure 8.2 Anthill three layers structure

As depicted in figure 8.2, high−level applications are allowed to directly interact with the Anthill infrastructure. Theoretically applications should not be aware of ants. What an application can do is strictly limited to instantiating and passing valid queries to their actual nest instance via the provided methods. An issued query may then be followed by the receipt of a response message. A class willing to receive responses associated with formerly issued queries, has to implement the *anthill.nest.ResponseListener* interface. The application proposed along with this thesis is a document sharing application for sharing technical reports among universities and researchers. We have decided to test the Anthill framework in such a field mainly for comparing our peer−to−peer system's performance to the ones exposed by nowadays file sharing systems. What we are more interested in is the behavior of the ant species used for providing the above service. In order to show how an high level application can easily interact with a nest, it is interesting to show the piece of source code in charge of passing requests to the Anthill infrastructure.

```
RequestId reqId = new RequestId( localNest.getNestId() );
Query query = new Query((String)searchByBox.getSelectedItem(),
              keywords );
QueryResults results = new QueryResults(
localNest,searchByBox.getSelectedItem().toString().trim(), keywords );
localNest.request( reqId, query, results);
```

Figure 8.3 Code issuing a search query

```
RequestId reqId = new jxtaimpl.ids.RequestId( nest.getNestId() );
String keys = file.getName()+" "+type+" "+desc;
nest.insRequest( reqId, keys );
```

Figure 8.4 Code issuing an insertion−event message

In both cases what the application does is simply instantiating an identifier for the query it is about to issue, and then the most appropriate query object, if required. The last step is calling one of the methods provided by the Anthill nest. This way applications have not to deal with ants and other related issues, since it is all transparently handled by the Anthill infrastructure.

8.2 TechRep: Technical Report Sharing Application

TechRep is an example application layered on top of the JXTAnthill Project, meant for sharing technical reports in a peer−to−peer fashion. The *TechRep* application presented along with this thesis is in its preliminary stage, and we expect to improve it in the future. The usage of this application will let us explore the possibilities offered by the synergic employment of the peer−to−peer architecture and the mobile−agents paradigm. This application's code resides in the *techrep* package. Figure 8.1 depicts the *techrep* UML Class Diagram. Besides the classes merely used for building this application's GUI, our interest has to be focused in the *TrBibReader* package, whose UML class diagram is depicted in figure 8.7.

8.3 Background: The BIB Syntax (CS−TR−V2.1)

Before going deeper into the detailed explanation of the *TrBibReader* package, it is necessary to introduce some of the concepts fundamental to the remainder of this chapter. At the moment, all the technical reports we want to share are accessible via ftp through the web. Each technical report is associated with a textual metadata describing the technical report itself. All these metadata have to be compliant with the syntax imposed by the current BIB version adopted, and are arranged into textual files. A sample BIB metadata is the following:

```
BIB−VERSION:: CS−TR−v2.1
         ID:: ncstrl.cabernet//BOLOGNA#UBLCS−99−10
      ENTRY:: June 10, 1999
ORGANIZATION:: University of Bologna (Italy). Department of Computer
               Science.
      TITLE:: Proceedings of the Workshop on Virtual Documents,
               Hypertext
            Functionality and the Web
     AUTHOR:: Milosavljevic, M.
    CONTACT:: <Maria.Milosavljevic@cmis.CSIRO.AU>
     AUTHOR:: Vitali, F.
    CONTACT:: <fabio@cs.unibo.it>
     AUTHOR:: Watters, C.
    CONTACT:: <watters@cs.dal.ca>
       DATE:: May 1999
      PAGES:: 51
  COPYRIGHT:: Department of Computer Science, University of Bologna,
               Italy. All rights reserved.
OTHER_ACCESS:: URL:ftp://ftp.cs.unibo.it/pub/techreports/99−10.ps.gz

ABSTRACT::

Collection of papers presented at the Workshop on Virtual Documents,
Hypertext Functionality and the Web.

END:: ncstrl.cabernet//BOLOGNA#UBLCS−99−10
```

Figure 8.5 bib file sample entry

In order to more easily query the resource manager, we want to translate all these bib entries into XML documents. The DTD of the documents we want to generate is the following:

```
<?xml version=”1.0” encoding=”us−ascii”?>
<!ELEMENT TrBib (Author+)>
<!ATTLIST TrBib BIB−VERSION CDATA #REQUIRED>
<!ATTLIST TrBib ID CDATA #REQUIRED>
<!ATTLIST TrBib ENTRY CDATA #REQUIRED>
```

```
<!ATTLIST TrBib ORGANIZATION CDATA #REQUIRED>
<!ATTLIST TrBib TITLE CDATA #REQUIRED>
<!ATTLIST TrBib DATE CDATA #REQUIRED>
<!ATTLIST TrBib COPYRIGHT CDATA #REQUIRED>
<!ATTLIST TrBib OTHER_ACCESS CDATA #REQUIRED>
<!ATTLIST TrBib ABSTRACT CDATA #REQUIRED>
<!ATTLIST TrBib  END CDATA #REQUIRED>

<!ELEMENT Author>
<!ATTLIST Author Name CDATA #REQUIRED>
<!ATTLIST Author Contact CDATA #REQUIRED>
```

Figure 8.6 DTD of the XML document used for representing bib items

8.4 Package *TrBibReader*

This package contains the classes needed for parsing a bib file and for generating an XML representation of its textual content. As figure 8.7 depicts, there are three classes:

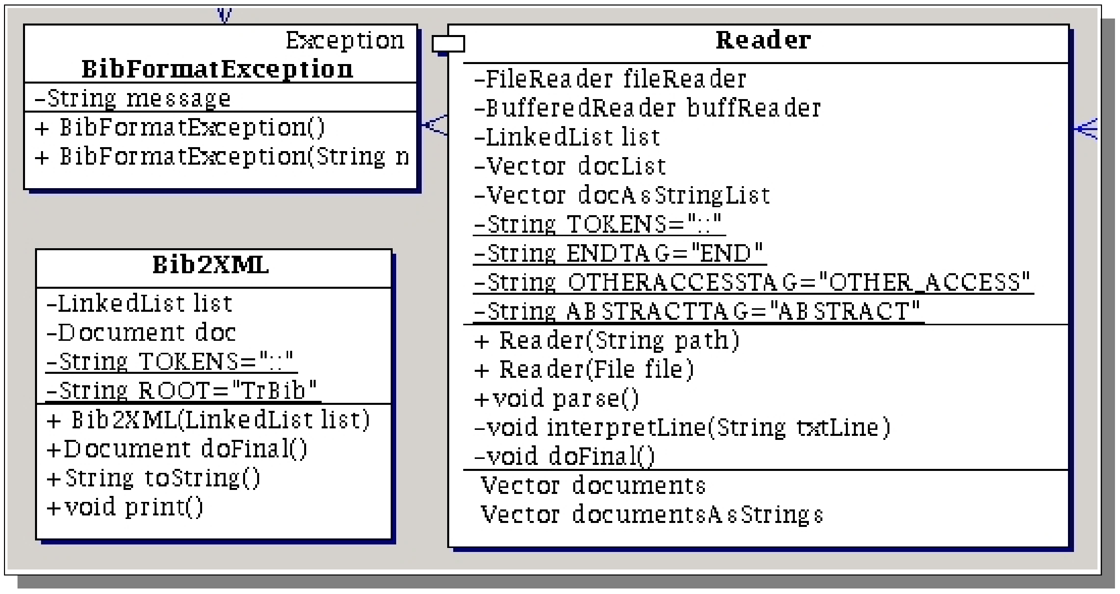


Figure 8.7 TrBibReader UML Class Diagram

- *TrBibReader.Reader*
- *TrBibReader.Bib2XML*
- *TrBibReader.BibFormatException*

8.4.1 Class *TrBibReader.Reader*

This class' constructor takes as input parameter a *File* object, or a *String* representing the complete path to the bib file describing the resources we want to share. After having instantiated a *Reader*, what we have to do is invoking its *public void parse()* method. This method will parse the bib file used to instantiate the *Reader* object and will store in a couple of *Vector* objects both the *org.w3c.dom.Document* and *String* representation of each entry. These two data structures can be later on retrieved invoking the following methods:

- *public Vector getDocuments()*
- *public Vector getDocumentsAsStrings()*

8.4.2 Class *TrBibReader.Bib2XML*

Each *Reader* instance, leverage on the *TrBibReader.Bib2XML* class for accomplishing its task. What this class does is translating a single bib item into an appropriate XML document. This class' constructor takes, as input patameter, a *LinkedList* instance. Each entry of this list corresponds to a bib tag along with its value. For example, given the bib entry depicted in figure 8.8, the first element contained into the list will be: `BIB−VERSION:: CS−TR−v2.1`

```
BIB−VERSION:: CS−TR−v2.1
         ID:: ncstrl.cabernet//BOLOGNA#UBLCS−99−10
      ENTRY:: June 10, 1999
ORGANIZATION:: University of Bologna (Italy). Department of Computer
               Science.
      TITLE:: Proceedings of the Workshop on Virtual Documents,
               Hypertext Functionality and the Web
     AUTHOR:: Milosavljevic, M.
    CONTACT:: <Maria.Milosavljevic@cmis.CSIRO.AU>
     AUTHOR:: Vitali, F.
    CONTACT:: <fabio@cs.unibo.it>
     AUTHOR:: Watters, C.
    CONTACT:: <watters@cs.dal.ca>
       DATE:: May 1999
      PAGES:: 51
  COPYRIGHT:: Department of Computer Science, University of Bologna,
              Italy. All rights reserved.
OTHER_ACCESS:: URL:ftp://ftp.cs.unibo.it/pub/techreports/99-10.ps.gz

ABSTRACT::

Collection of papers presented at the Workshop on Virtual Documents,
Hypertext Functionality and the Web.


END:: ncstrl.cabernet//BOLOGNA#UBLCS-99-10
```

Figure 8.8 bib file sample entry

Once the *Bib2XML* class has been instantiated, invoking the *public org.w3c.dom.Document doFinal()* method we obtain the XML representation of the bib item, shaped as a DOM (*Document Object Model*). The *String* representation of the same XML document, can be obtained through the *public String toString()* method.

For example, the XML representation of the bib entry depicted in figure 8.8 is represented in figure 8.9.

```
<?xml version="1.0" encoding="UTF-8"?>
<TrBib
      BIB-VERSION="CS-TR-v2.1"
      ID="ncstrl.cabernet//BOLOGNA#UBLCS-99-10"
      ENTRY="June 10, 1999
      ORGANIZATION:: University of Bologna (Italy). Department of Computer
                     Science."
      TITLE="Proceedings of the Workshop on Virtual Documents, Hypertext
            Functionality and the Web"
      DATE="May 1999"
      PAGES="51"
      COPYRIGHT="Department of Computer Science, University of Bologna,
                Italy. All rights reserved."
      OTHER_ACCESS="URL:ftp://ftp.cs.unibo.it/pub/techreports/99-10.ps.gz"
      ABSTRACT="Collection of papers presented at the Workshop on Virtual
               Documents, Hypertext Functionality and the Web."
      END="ncstrl.cabernet//BOLOGNA#UBLCS-99-10">
      <AUTHOR
            NAME="Milosavljevic, M."
            CONTACT="Maria.Milosavljevic@cmis.CSIRO.AU"/>
      <AUTHOR
            NAME="Vitali, F."
            CONTACT="fabio@cs.unibo.it"/>
      <AUTHOR
            NAME="Watters, C."
            CONTACT="watters@cs.dal.ca"/>
</TrBib>
```

Figure 8.8 XML representation of the bib entry depicted in figure 8.7

8.4.3 Class *TrBibReader.BibFormatException*

This class simply defines an exception the *TrBibReader.Reader* instance will throw if the provided input file is not compliant with the recognized bib format version (CS−TR−V2.1 at the moment).

8.5 Resource Sharing Ant Species: Gnutant

At the moment, the technical report sharing application employs a generic virtual ant for accomplishing its tasks. In the future we have planned to develop a more sophisticated and specialized ant species that will be able to handle and understand the content of the XML documents describing the shared resources. This is needed to increase the number of successful searches. What follows is the description of the virtual ant species employed at the moment along with the *techrep* application: *Gnutant*.

Gnutant is an artificial ant species devised for building a file−sharing application. Ants belonging to this ant species are in charge of executing two main tasks:

- Building and maintaining a distributed file index.
- Performing searches in the Anthill network.

Each resource is associated with some *meta−data* including a set of textual *keywords* and a unique *file identifier*. This file identifier is the same for all the potential replicas of the same resource. This approach is needed to provide faster file downloads, by requesting disjoint fragments of the file from multiple location (not yet supported by the implementation). The distributed file index is used by Gnutant ants for routing purposes. This index is made up of entries that associate a set of next hops in the Anthill network with the hash value of a keyword. Each hashed keyword is computed using the Secure Hash Algorithm (SHA) to obtain a 160 bit value. When an ant performing a search reaches a nest, it will inspect the routing storage using the keywords it is assigned to. If an exact match is found, the ant will select one of the entries in the set of next−hop nests associated with the matching hashed keyword; otherwise, it will select a nest associated with the "closest" hashed keyword. This notion of closeness is fundamental to Gnutant's routing scheme. Nests associated with a particular hashed keyword in a routing storage will tend to receive more requests for keywords similar to it. Thus they will tend to garner URLs referring to resources all exposing similar hashed keywords, giving life to a clustering phenomena that will improve the search performance over time, enabling ants to quickly find the relevant region in the nest network [BMM−09−01]. The Gnutant ant species envisages three task−specialized ant types:

- *InsertAnt.*
- *SearchAnt.*
- *ReplyAnt.*

InsertAnt. Ants belonging to this category are in charge of updating the distributed index upon the insertion of a new resource into an Anthill nest.

SearchAnt. SearchAnts are generated by nests to satisfy users requests for resources. These ants exploit the routing information to determine the shortest path to files matching the user requests. SearchAnts are associated with a TTL (Time−to−Live) value: upon reaching its TTL, the ant will return to its originating nest backtracking the followed path and updating the distributed index to reflect its findings.

ReplyAnt. These ants are generated by SearchAnt instances once a resource has been found. They are meant to return immediately to their home nest bringing there the set of useful results. This way the SearchAnt instance may keep on following new paths looking for more resources.

8.6 Gnutant Ant Species: Simulation Results

The Gnutant ant species has been tested using the Anthill simulation environment for comparing its real behavior to our expectations. The simulated scenario consisted of a 2000−nodes static network. These preliminary tests didn’t take into account node crashes, but tests in a dynamic scenario have already been planned. The Gnutella network has been monitored over a period of 30 minutes for collecting a set of 10000 real queries. These queries have been used for randomly scatter resources into our simulated Anthill network, before running the simulation. This way, potentially all of the queries could have been satisfied. Each routing storage has been initialized with randomly generated SHA keys, thus making ants move at random in the beginning. Ants have been assigned to a Time−To−Live (TTL) value equal to 100 hops and, after the insertion phase, the simulation has been run ten times issuing 20000 search requests per session. Routing, resource and URL storage have been set with a capacity of 16, 16, and 64 respectively. Upon receiving a successful response, has been simulated the download of 10% of the matching files, followed by the execution of an InsertAnt for each downloaded file. This task is necessary for keeping the distributed index as much updated as possible.

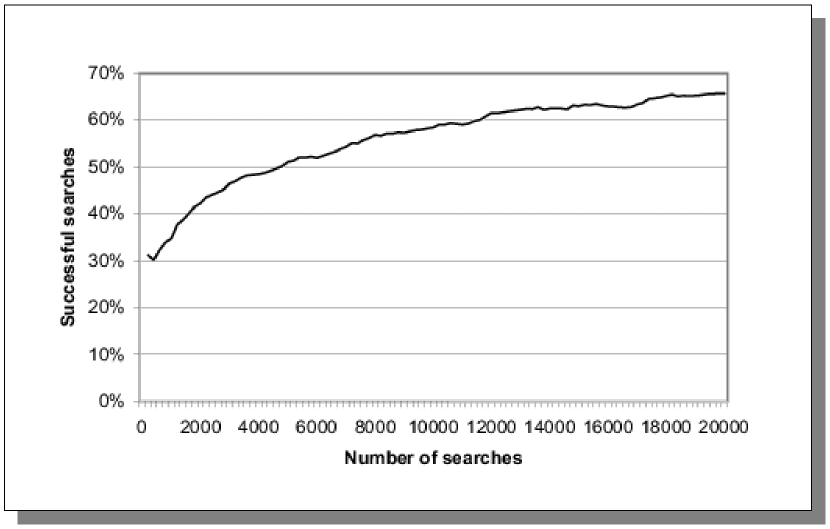


Figure 8.9 Gnutant Simulation Results: successful searches increase as number of searches grows up [BMM−09−01].

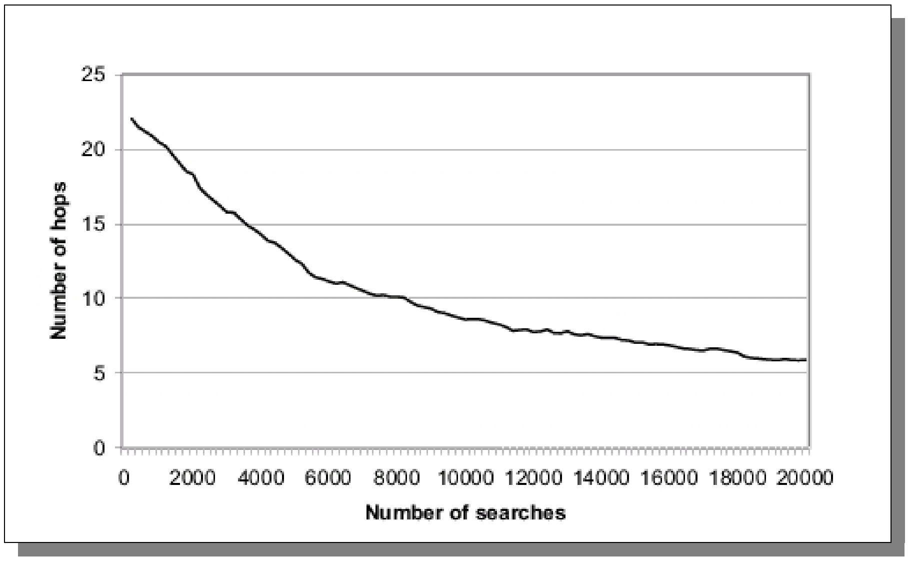


Figure 8.10 Gnutant Simulation Results: number of hops decreases as the number of searches grows up [BMM−09−01].

As we can see in figure 8.9, the number of successful searches increases as the number of searches grows up, while the number of required hops in the Anthill network goes down (figure 8.10). The system seems to converge towards a 65% success rate for searches and approximately six hops for the average search depth. These data confirm our expectations: the overall performance of the system increases with the growth of interaction among the system's components, the nodes.

Appendix − UML Sequence Diagrams

Bibliography

[BMM−09−01] Özalp Babaoglu, Hein Meling, Alberto Montresor. Anthill: A Framework for the Development of Agent−Based Peer−to−Peer Systems. (2001)

[PJVN] Bernard Traversat, Mohamed Abdelaziz, Mike Duigou, Jean−Christophe Hugly, Eric Pouyoul and Bill Yeager. Project JXTA Virtual Network. http://www.jxta.org (February 5, 2001)

[SPJ] Sun Microsystems, Inc. Security and Project JXTA. http://www.jxta.org (January 23, 2002)

[POBLANO] Rita Chen and William Yeager, Sun Microsystems, Inc. Poblano A Distributed Trust Model for Peer−to−Peer Networks. http://www.jxta.org (2002)

[PJTO] Li Gong, Sun Microsystems, Inc. Project JXTA: A Technology Overview. http://www.jxta.org (April 25, 2001)

[PJOIC] Sun Microsystems, Inc. Project JXTA: An Open, Innovative Collaboration. http://www.jxta.org (April 25, 2001)

[PJJFPG] Sun Microsystems, Inc. Project JXTA: Java™ Programmer's Guide. http://www.jxta.org (December 7, 2001)

[SOSI] Eric Bonabeau, Guy Theraulaz, Jean−Louis Deneubourg, Serge Aron, Scott Camazine. Self−Organization in Social Insects. Santa Fe Institute. (February 20, 2002)

[POKA] Eric Bonabeau, Guy Theraulaz, Vincent Fourcassié, Jean−Louis Deneubourg. The phase−ordering kinetics of cemetery organization in ants. Santa Fe Institute. (February 20, 2002)

[P2PAM] Alberto Montresor. The Anthill Project − Part 1: Introduction to Peer−to−Peer. http://www.cs.unibo.it/montreso (2001)

[GC] **Gartner**Consulting, GartnerGroup. The Emergence of Distributed Content Management and Peer−to−Peer Content Networks. (January 2001)

[SATH] Eric Korpea, Dan Werthimer, David Anderson, Jeff Cobb and Matt Lebofsky. SETI@home: Massively Distributed Computing for SETI.

[WIAWIN] Clay Shirky. What is P2P... And What Isn’t? http://www.oreillynet.com/pub/a/p2p/2000/11/24/shirky1−whatisp2p.html (February 2002)

[P2P4A] Andy Oram. Peer−to−Peer for Academia. http://www.oreillynet.com/pub/a/p2p/2001/10/29/oram_speech.html (February 2002)

[DST] Nelson Minar. Distributed Systems Topologies: Part 1−2. http://www.oreillynet.com/pub/a/p2p/2001/12/14/topologies_one.html http://www.oreillynet.com/pub/a/p2p/2001/12/14/topologies_two.html (December 12, 2001)

[TCBON] Gary William Flake. The Computational Beauty of Nature, Computer Exploration of Fractals, Chaos, Complex Systems, and Adaption. A Bradford Book, The MIT Press. (1999)

[CNET−01] Larry Peterson and Bruce S. Davie. Computer Networks, Second Edition. Edited by Morgan Kaufman. (2000)

[DS] Distributed Systems, Second Edition. Edited by Sape Mullender. Addison−Wesley. (1998)

[CMS] Content Management Service. http://www.jxta.org (May 6, 2002)