TR - H - 145

Tierra Network Version

Agnés CHARREL

1995. 4. 27

(1995.1.31受付)

ATR人間情報通信研究所

〒619-02　京都府相楽郡精華町光台2-2　☎ 0774-95-1011

ATR Human Information Processing Research Laboratories

2-2, Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02 Japan

Telephone: +81-774-95-1011

Facsimile: +81-774-95-1008

Institut National des
Télécommunications

Advanced Telecommunications Research Institute

Rapport de stage
de troisième année

Technical Report

Tierra Network Version

Agnès Charrel

August 1994 - January 1995

Table of Contents

0-Introduction

This work was done in the Department 6 (Evolutionary Systems) of ATR Human Information Processing Laboratories, Kansai Science City, Japan, under the direction of Professor Thomas S. Ray.

The subject of this work was to help implement a network version of an existing software package called Tierra.

Tierra is a program that provides the necessary environment for Artificial life creatures to live in. And the ultimate goal of the project was to enable Tierra to run over the Internet, the giant computer "internetwork" that spans the world.

This would allow the creation of a network-wide biodiversity reserve for digital organisms (see [9]), where they would be left to evolve freely.

The work to be done was two-fold.

First we had to add to Tierra the communication routines that would allow it to run over a network, and more specifically a network using the TCP/IP protocol suite.

Secondly we had to provide the creatures the means to use the capabilities of a networked version of Tierra.

These are two different tasks, due to the nature of Tierra itself.

The digital organisms in Tierra are self replicating programs. But they are not running on any real computer. Instead they run on a virtual computer with a reduced instruction set, whose hardware and operating system are simulated by the Tierra software. Tierra in turn runs on any of a variety of hardware machines (from IBM PC under MS-DOS to mainframes under Unix).

In the same way, the network version of Tierra is a virtual computer network running on a real computer network. Tierra is the interface between the underlying hardware and the creatures, which are only aware of a simplified networked world, simulated by the Tierra software.

As a consequence, the implementation of the new network functionalities required learning about both Tierra itself, and the workings of computer networks, and more precisely the Internet.

I-Tierra

To begin with, understanding Tierra requires knowing a few things about its background, Evolutionary Computation and more precisely about Artificial Life.

In this chapter we will quickly review both domains, and show the situation of Tierra in its field. We will then explain what are the goals of the Tierra project, and why a network version of Tierra was needed.

I.1-Evolutionary computation

I.1.1-Definitions

As stated in [1], evolutionary computation consists of methods using "computational models of evolutionary processes as key elements in the design and implementation of computer-based problem-solving systems".

An **evolutionary algorithm** (as in **EA**) works on a "population", which is a set of solutions to a problem encoded in a form allowing manipulation by a computer. This population is evolved through reproduction and selection of individuals, until a satisfying solution to the initial problem is found.

An EA usually begins its work with a population initialized at random (a set of preliminary solutions given by the user, or obtained through another method could also be used). The EA then selects the parents for the next generation, reproduces the chosen individuals, and (if necessary) selects part of the offspring to compose the new population (most of the EAs work with a stable population, meaning that the number of individuals does not vary in the time).

The EA goes through these different operations in a loop until the population offers a satisfying solution or set of solutions.

As can be seen above, the operations of selection and reproduction compose the heart of the EA. The details may vary from one model to another, but the general principle is always the same.

Selection is done in two steps. The EA first evaluates the **fitness** of each individual (the fitness and the function used to measure it are defined by the user, along with the representation of individuals, as parameters of the EA). The different individuals are then classified, and only the ones with the best fitness are used for the next stage. The others are discarded.

Quite logically an EA looks for the "best" (according to predefined criteria) solutions to the problem.

But selection is meaningless if the individuals remain the same generation after generation. An EA must create at least a few new solutions at every generation, and then select for the best ones (which may or not include the new ones).

Reproduction uses recombination or mutation (inspired from the biological mechanisms bearing the same name) to create individuals different from the parents.

Recombination requires at least two parents. Children are created by combining part of the "genome" (computer representation of a solution) of each parent to form a new individual different from them. The parameters for recombination are the point(s) of crossover (position in the genome of parents from which the sections of genome will be swapped), which may be always the same, or, better, chosen according to a probabilistic function.

The other operation for reproduction is **mutation**, which modifies part of the genome of a parent to create a child. The modification is controlled by a probabilistic function, which can be as simple as flipping a bit in the genome with a (low) fixed probability, or as complicated as a set of different and possibly quite complex functions for each part of the genome.

An EA can use mutation or recombination or both to create new individuals at each generation, thus exploring the space of solutions without perturbing too much the set of existing solutions. Because in every cycle the new individuals with a fitness inferior to the others' are discarded, the average fitness of the population grows steadily.

As will be seen in section I.3, different models of EA exist. They differ by the representation of individuals, the selection mechanism, and the operator(s) for reproduction. These choices are often conditioned by the nature of the problem being treated.

I.1.2-Why?

The first section explained what evolutionary computation is. We saw that they are used as problem solvers, but computers were solving problems long before the apparition of EAs, and a method that looks for solutions more or less at random does not seem so efficient compared to other computing methods...

The main reason why evolutionary computation models appeared is the lack of adequation between traditional computing methods and many of the problems that currently need to be solved.

These problems are very complex (they involve a huge number of parameters), and deal with "real-world" phenomenons which are either described by a complicated model (non-linear, non-continuous...) or not modelized at all (usually because the phenomenons are not fully understood).

To try and solve these problems, we have the "traditional" computing methods. They usually require a lot of restrictive hypothesis to be respected, among which linearity and continuity are proeminent.

In cases when hypothesis are not too restrictive, then calculation becomes very complex, and needs a huge amount of raw computer power to execute in a reasonable amount of time.

Even worse, the computational costs for traditional methods have a bad tendency to grow exponentially with the number of parameters of the problem.

Evolutionary computation introduce an alternative to traditional methods for very complex problems. It does not guaranty the exact solution to a given problem, but provides a good approximation in a finite time and within practical computational costs.

However, EAs are weak methods for solving problems. Some of their blind spots are exposed in the following examples.

If an EA finds a local maximum (of the fitness function) in the space of potential solutions, surrounded by an area where all the individuals have a very low fitness, the perturbations introduced by the reproduction operators may not be enough for a new individual to reach the real maximum and pull the population toward this new attractor.

The situation is even worse if the maximum is also a very steep peak in a rather "flat" area of low fitness.

Another problem appears when more than one solution exists. If these solutions are far away, and (once again) are separated by areas of low fitness, the EA may settle around one solution and never discover the other ones.

Despite their weaknesses, EAs provide a good alternative to traditional methods when these are too costly (either in time or computer power) or impossible to use (cases when there is no comprehensive model for the problem).

I.1.3-Current models

The principles behind evolutionary computation were discussed as early as in the 50's, but the first models appeared in the late 60's and early 70's. These are "evolutionary programming" (Fogel, 1966, see [4]), "evolution strategies" (Rechenberg, 1973, see [5]) and "genetic algorithms" (Holland, 1975, see [6]).

These three models were very different at the time of their creation. After twenty years of evolution, they remain very much different, even if united under the generic term of EAs. They still compose the basis of the work in evolutionary computation (along with other models later derived from them).

Here is a short description of the three models, showing the possible variations in the generic EA presented in the first section. It should be noted that EAs are a generalization of these three models, and were formalized long after they were first created.

I.1.3.1-Evolutionary programming (EP)

The representation of individuals in EP is traditionally problem-dependent. If the problem deals with real-valued vectors, the individuals in the population will be real-valued vectors. A traveling salesman problem will use ordered lists to represent individuals, and so on.

EP is often used for optimization problems.

Figure 1.1 shows the algorithm for a typical EP.

```
t = 0;
initialize population P(t)
evaluate P(t)
until (done) {
   t = t + 1;
   parent_selection P(t);
   mutate P(t);
   evaluate P(t);
   survive P(t);
}
```

fig. 1.1: evolutionary programming algorithm

Supposing that the population amounts to N individuals, the parent selection chooses all of them. Reproduction produces one offspring for each parent through mutation only. Parents and children (2N individuals) are then evaluated. N individuals survive to compose the population at time t+1. The choice of surviving individuals uses a probabilistic function based on fitness, meaning that individuals with a high fitness are more likely to survive.

The mechanism of mutation can be complex and always depends on the representation chosen for the individuals. Each variable within an individual may have a different, adaptive mutation rate. Recombination is not generally used by EP, mostly because the form of mutation used can provide individuals with perturbations akin to those recombination would introduce.

I.1.3.2-Evolution strategies (ES)

ESs usually deal with real-valued vector representations (the first applications for ESs were hydrodynamic optimization problems).

Figure 1.2 shows the algorithm for a typical ES.

```
t = 0;
initialize population P(t)
evaluate P(t)
until (done) {
   t = t + 1;
   parent_selection P(t);
   recombine P(t);
   mutate P(t);
   evaluate P(t);
   survive P(t);
}
```

fig. 1.2: evolution strategy algorithm

The parent selection is done at random. The parents thus chosen produce children (more than N if N is the size of the population) through recombination. The offspring is further perturbed by mutation. The ES then chooses N individuals to form the new population, either amongst the children or amongst the parents and children. Survival in ES is deterministic, and selects the individuals with the highest fitness.

Like in EP, each variable in an individual can have an adaptive mutation rate. However, recombination is important in the working of an ES, especially to evolve the mutation along with the individuals.

I.1.3.3-Genetic algorithms (GA)

GAs traditionally use bit-strings to represent individuals. This method provides a more domain-independent approach.

GAs are often used as optimizers, but their capabilities are more general.

Figure 1.3 shows the algorithm for a typical GA.

```
t = 0;
initialize population P(t)
evaluate P(t)
until (done) {
   t = t + 1;
   parent_selection P(t);
   recombine P(t);
   mutate P(t);
   evaluate P(t);
   survive P(t);
}
```

fig. 1.3: genetic algorithm

Parent selection is done using a probabilistic function based on fitness (the better the fitness, the more likely for an individual to be selected as a parent). N children are created by recombining the N parents. They are mutated and survive, replacing their parents in the population.

In GAs recombination is the most important reproduction operator. Mutation is more of a background operation, and simply flips bits in the individuals with a low (fixed) probability. (Some researchers even argue that mutation is not necessary in GAs as recombination can produce the same results).

I.2-Artificial Life

Artificial life was defined by Langton (in [3]) as "the study of man-made systems that exhibit behaviors characteristic of natural living systems".

There are two goals behind the conception and study of artificial life.

The first one is to try and understand more about life itself. It is possible for a biologist to study life-as-it-is, by observing the organisms present around us. But biology is very much limited to observation. When a physicist might observe the world, create a theory to explain its workings, and then begin making experiments, modifying the conditions to see if the theory remains valid, a biologist has very few possibilities to experiment with life. Furthermore, even observation is limited to the life we know. A biologist cannot observe life as it existed 10 million years ago, or as it will exist in 10 million years on Earth, for example. Fossils are a poor substitute to live specimen. To be fully able to study life, a biologist would have to be able to create worlds and observe life on them during a very long time.

Creating new worlds with different life forms is perhaps not so impossible as it may seem. A computer simulation of these other worlds would be easy to create and to manipulate (at least much more so than a set of new planets). And life could be generalized to other forms than the one we know (based on carbon and water), further enlarging the field of study for biologists.

The beginnings of artificial life led to a lot of questions about the nature of life itself. To create life, quite often in a medium rather different from the usual one (software for example), one must be able to recognize life.

A list of properties that are characteristic of life as we know it was established (see [3]), but the boundary between life and "not-life" remains imprecise. It is hoped that further research and experimentation with artificial life forms will bring a more precise, more comprehensive definition of life.

The second goal of artificial life is somewhat more practical. It would be an attempt to use the new (life) forms created by artificial life for technical applications.

These applications may exist in very different fields. Artificial life has been applied to three main media: “wetware”, or carbon-based life-forms, hardware, and software.

The practical applications for wetware are the most evident. Through genetic engineering many new species of plants and animals (to a lesser degree) have been created by scientists. Amongst hardware applications of artificial life was a feasibility study for the NASA about the possibility of building self-repairing and relatively independent space probes. Software makes for easy manipulation. Possibilities for this medium range from the optimization of existing software to the creation of an artificial brain.

I.3-Tierra

I.3.1-The beginnings

In 1989 Tom Ray, an evolutionary biologist with an interest in computer science, decided to try and combine both disciplines to create a new world, that would allow him to observe evolution.

Because evolution is all around us, but works at a very slow pace. The average human life span is as nothing compared to the hundreds of millions of years nature needed to get from the first primitive life form to the many complex creatures currently living around us. This makes the actual observation of evolution rather difficult.

The idea was to write a small self-replicating program, similar in principle to the first living creature on Earth. This program would then be allowed to run. Every time the program would reproduce, there would be a small probability for a mutation to occur, producing a change in the replicated creature. This is the way life in all its complexity developed on Earth. It was hoped that the same complexity would develop relatively quickly inside a computer, at the fast pace allowed by modern micro-processors.

This was in fact an evolutionary approach to artificial life, which was at that time emerging as a new discipline. Tom Ray contacted Chris Langton, and was invited to visit the Artificial Life group at Los Alamos National Laboratories in October 1989. He discussed his ideas with the different members of the group.

Most of them were skeptical, as they objected that mutation was not a good thing to use on standard computer languages. These are too fragile, too “brittle” for such a treatment. Mutation done on computer code would be almost sure to produce junk. It would be next to impossible for the first self-replicating creature to mutate into anything but an incorrect program.

Tom Ray took the argument to heart, but nonetheless went on to try and implement his idea. The result was, of course, Tierra.

I.3.2-Basic principles

I.3.2.1-A reduced instruction set

The basic problem with the idea of mutating computer code is, as we said above, the brittleness of computer languages. And that does not even take the high-level languages (such as C, Pascal or Fortran) into account.

For a common modern processor the assembler language uses 32-bit words to encode one instruction. So there are 2^{32} possible instruction words. The brittleness stems from this point. If an instruction in the code of the creature is modified (a mutation would typically flip one bit in the 32-bit word), the probability for this instruction to be meaningful (or even harmless) in the context of the rest of the code is next to null, even when considering that more than one instruction can be meaningful.

However, mutation seems to be getting somewhere when applied to the DNA of organic life forms. I am not trying to say that a mutation is always a good thing for the mutated creature. On the contrary, the probability for a mutation to be harmless is low, and the probability for it to create an improvement over the non-mutated creature is even lower. However, these events are far from being as unlikely as the mutation of a good program into a better program.

So what is the difference?

The genetic language for organic creatures is based on an alphabet of four characters: the nucleic acids (C, G, A, T). These are combined in threes to encode one amino acid. So 64 (4^3) amino acids can potentially be encoded by the four nucleic acids. However, the encoding is redundant, and the 64 possible combinations of nucleic acids are mapped to only 20 actual amino acids.

An organic mutation exchanges one nucleic acid for another. The modification would modify one amino acid, getting the new one among twenty possibilities (the probability for getting the same one being non negligible).

For a mutation on computer code, a mutation on one bit would choose the mutated instruction in a pool of more than four billions (2^{32}) instructions.

The conclusion is that "the likelihood of finding something useful when you must swap among four thousand million objects seems much less that when you must swap among only twenty objects" [7].

So the first thing to do was to reduce the size of the instruction set to something nearer to 20. The new instruction set would consist of only 32 instructions (2^5).

A processor does not perform many different operations. Most of the many instruction codes used by a traditional (CISC) computer deal with the many possible combinations for operands. The new instruction set requires all the machine instructions to operate on numbers contained in or pointed to by the registers of the CPU. And there will be no floating point operations.

As a consequence the number of instructions needed to run the machine decreases drastically, and 32 of them are quite enough.

I.3.2.2-Branching and looping

Another mechanism borrowed from biology was added to the computer being built. For this is what all of this is about, building a new computer, more adapted to self-reproducing programs submitted to mutations.

Sometimes a piece of code needs to interact with another one in a distant region of memory. This is when a call or jump instruction appears in the code. A computer must then specify the address of the next piece of code to be executed. So the program must always know the precise address of the destination in the memory.

But in organic creatures, interactions between molecules in the cell do not require one molecule to know the precise location of another molecule. Instead, elements in a cell present a 3D pattern to which other molecules fit in a lock-and-key fashion. Diffusion brings them together, and complementary shapes allow them to interact (or not).

A similar mechanism was introduced to the code: addressing by complementary templates. Instead of specifying a jumping address, the code gives a pattern, and the jumping instruction goes through the memory looking for the reverse pattern. Once it is found, execution continues from this point.

The new instruction set counts two nop (no operation) instructions: nop0 and nop1. The patterns (templates) used by the branching and looping instructions are implemented as a succession of these nop instructions. For example, if the pattern used as a starting point for a jump is {nop0, nop0, nop1, nop0}, then the jump instruction will look for a {nop1, nop1, nop0, nop1} after which it will resume execution.

I.3.2.3-The reaper

A self-replicating program executes in an endless loop, creating a self-replicating child every time it goes through the loop. However, the resources inside a computer are limited. What happens when the self-replicating programs have completely filled the memory? They cannot run anymore, because they cannot allocate enough memory to spawn a child.

To avoid blocking evolution completely in a very short time, a mechanism was introduced to free memory when there is none available and when a creature (a self replicating program) tries to replicate. This mechanism was called the reaper, and added death to the Tierran universe.

When a creature is created, it enters the reaper queue. It is basically a FIFO (first in first out) queue. When Tierra needs to allocate memory and the soup (memory space dedicated to the creatures) is full, the first creature in the queue is erased from memory, and the memory space thus freed can be reallocated. Using a FIFO queue simply means that the oldest creature is the first one to die, which is quite logical.

In fact, the reaper queue is not quite a FIFO queue. A creature can jump forward in the queue if it misbehaves, i.e. if it raises error flags too often when executing. This gives a shorter life span to creatures crippled by mutations, making more room for viable creatures.

The different ideas exposed above seemed to solve part of the problems raised by the use of mutations on self-reproducing code. But would they be enough to create evolution?

I.3.3-Results and further developments

I.3.3.1-Implementation and first run

Tom Ray created a new computer using the ideas above. This computer had a 32-instruction instruction set, and used templates for looping and branching.

Of course, nobody went to create an actual chip encoding this instruction set in silicon. Instead, as is actually done when testing a new computer architecture, Tom Ray wrote a piece of software that simulated the required computer, and called it Tierra (Earth in spanish).

The first test was done by running, quite appropriately, a (80-instruction) self-replicating program. Here is what happened, as remembered by Tom Ray (in [7]):

"I never intended that this virtual computer and my first rudimentary self-replicating program should be anything more than a starting point. I expected to spend years modifying the design of the computer, and testing even more sophisticated self-replicating programs on it. My plans were radically altered by what actually happened on the night of January 3, 1990, the first time that my self-replicating program ran on my virtual computer, without crashing the real computer that it was emulated on."

"All hell broke loose. The power of evolution had been unleashed inside the machine, but accelerated to the megahertz speeds at which computers operate [...] I was back in a jungle [...], but this time a digital jungle."

I.3.3.2-A rich ecology

This section will briefly describe part of the creatures that appeared, fought for life, and died in this digital jungle. As we will see, most of the adaptations brought by evolution are adaptations to the other creatures present in the soup, rather than adaptations to the very simple environment created by Tierra.

At the beginning was the ancestor. It was 80-instruction long, and endlessly replicated itself in memory. As it began filling the soup, it became a new resource of the environment. Some new creatures appeared in memory, a mutation having robbed them of part of their code. But the information they needed to reproduce was all around them, in the code of the "normal" creatures.

The first parasites were born. Because they were only half as long as the ancestor, they only needed half the time to reproduce. They were much faster. And they quickly invaded the soup.

However, they could not survive without their hosts because they needed part of the code in the host in order to reproduce. Every time they began dominating the soup, running the hosts to extinction, they died as quickly as they had come, starved for information.

Their dying allowed some breathing space for the hosts, which were able to reproduce again, thus adding to the soup the information needed by the parasite. This led to a stable oscillation between a soup dominated by parasites and a soup dominated by hosts.

This stability was short-lived. Hosts evolved ways to protect themselves from parasites, and parasites evolved ways to circumvent these protections.

This evolutionary race stopped when a new species of hosts appeared. These hosts had learned how to use the parasites to boost their reproduction, by tricking them into replicating their hosts. Because the tricky hosts had a complete genome, they did not really need another creature to help them reproduce. They simply got a bonus every time a parasite tried to use them. They were untouchable, and they invaded the world. After a while, they were the only thing left in the world.

Being alone in the soup, they evolved into social creatures, and became inter-dependent. The new creatures could only reproduce when gathered into clusters of close relatives. This cooperation implied trust, and shortly after cooperation appeared, a new brand of parasites evolved. These cheaters inserted themselves into a cluster of related creatures, and "when the trust was passed to them, they violated it" ([7]). They used a mechanism similar to the one the tricky hosts had evolved, and tricked the social creatures into replicating their parasites.

A new evolutionary race had begun...

Along the way of evolution, some creatures had discovered sex. When Tom Ray tried to turn off mutation to stop evolution for a while and observe the different creatures, evolution kept going. The creatures were mingling parts of their genome to create offspring with a code different from their parents. Evolution was under way and unstoppable..

I.3.3.3-Toward a Cambrian explosion of diversity?

It took Tom Ray two years to analyze what happened in the first run of Tierra. A rich ecology of creatures appeared, but he hopes for more in the future.

The ultimate goal would be the apparition of a virtual "Cambrian explosion of diversity".

The "Cambrian explosion of diversity" occurred 600 million years ago, when the first multi-celled creatures appeared on Earth. It took a long time for nature to get there: nearly three billion years. But once this stage was reached, the complexity of life increased exponentially in a short time, finally creating life-forms with nervous systems capable of coordinating sophisticated behavior.

"Evolution of complexity occurs in the context of an ecological community of interacting evolving species. Such communities need large complex spaces to exist. A large and complex environment consisting of partially isolated habitats differing and occasionally changing in environmental conditions would be the most conductive to a rapid increase in diversity and complexity."

"These are the considerations that lead to the suggestion of the creation of a large and complex ecological reserve for digital organisms. Due to its size, topological complexity, and dynamically changing form and condition, the global network of computers is the ideal habitat for the evolution of complex digital organisms" ([9]).

This global network of computers is, as you might have guessed, the Internet.

II-The Internet

The Internet is best known as the biggest computer network in the world. It allows its users to communicate with computers all around the world, through such applications as electronic mail, remote login or file transfer.

Its size is what made it most interesting to our project. With a total number of 3,212,000 computers connected (this figure was compiled in July 1994, and is now probably out of date, since the growth rate of the Internet was then estimated at +81% a year), the Internet could provide a lot of computing power, drawn from the unused cycles in most of these machines, but also an environment of maximum complexity.

Maximum complexity because, contrary to what it appears to be from the user's point of view, the Internet is not a computer network. It is rather, as its name suggest, an internetwork, which allows hosts on roughly 46,000 different networks to exchange information.

What makes the Internet possible is the TCP/IP protocol suite. It provides the necessary interface between heterogeneous network technologies and the application programs which are only the visible tip of the Internet iceberg.

The network version of Tierra being such an application program, learning how to interface it with the TCP/IP protocols was most important. Furthermore, we will later see how knowledge of the workings of the protocols themselves, beside being helpful for the network programmer, can be used to design more efficiently the interface between the Tierran creatures and Tierra itself.

II.1-The internetworking concept

II.1.1-A brief history of the Internet

Since the Internet was created and developed along with internetworking theory and technologies, its history perfectly illustrates the birth and evolution of the internetwork concept.

The first incarnation of the Internet was the ARPANET. It was funded by the Pentagon's Advanced Research Projects Agency both as a testing field and a means of communication for scientists working on packet switching technology across the country.

The Department of Defense was then trying to come up with a solution to communications in a post nuclear war situation. A packet switching network, which could keep running even after half its nodes had been blasted away, was that solution.

So in 1969, a first version of such a network, the ARPANET, was up and running with four switching nodes.

The protocols necessary to run such an internetwork were developed in the seventies, and the ARPANET grew with them, linking more and more sites, mostly research laboratories.

In 1979 the Internet Control and Configuration Board was created. Its goal was to loosely coordinate the ongoing research on internetworking technologies and to monitor the growth of the connected Internet, which had by that time begun to outgrow the ARPANET.

In 1980, the ARPANET switched from its original protocol, NCP, to the more sophisticated protocol suite called TCP/IP, and became the backbone of what had become the Internet.

In 1983, the ARPANET was split. One part, MILNET, reverted to the DOD and military communications. The other one, retaining the name ARPANET, was dedicated to the communication needs of the research community.

The protocol standards for TCP/IP were available to anybody, and the DARPA encouraged their linking to the popular BSD UNIX operating system, making them easier to use by application programmers. As a consequence, more and more organizations and universities were choosing TCP/IP as a network protocol suite, and hooking their existing networks to the Internet.

In 1984, the National Science Foundation joined the adventure. It began an ambitious program of expansion, in order to allow more scientists to get on the Internet.

In 1986, the NSFNET, the new long haul backbone linking the NSF supercomputers around the United States by the most advanced technologies available, was tied to the ARPANET.

The Internet began growing more and more quickly. To keep up with its growth, the NSFNET was upgraded and expanded in 1988, and again in 1990. The ARPANET officially expired in 1989, victim of its own success.

The ICCB, which had been reorganized and renamed as the Internet Activities Board in 1983, was again reorganized in 1989, to follow the changes brought by the evolution of the connected Internet. The IAB is now divided into two parts, the Internet Research Task Force and the Internet Engineering Task Force.

And it is mostly trying to keep up with the exploding growth of the Internet, by coming up with the technologies necessary to keep it running smoothly.

II.1.2-Internetworking

The history of the Internet explained how the Internet evolved along with its protocols. But it does not explain why an internetwork, instead of a specialized network, was needed by the Department of Defense.

The DOD was looking for a technology able to link all of its preexisting supercomputers and computer networks, using varied media for data transmission (ranging from conventional networking technologies to radio waves). This was a complex problem.

Networks are independent units. They consist of some computers, the hardware necessary to move information between them (lines, interfaces...), and some software to manage the links. This makes the physical and logical format of the data circulating on a network completely hardware dependent.

There is no simple solution to this problem, no possible dream of a universal standard for network hardware. The needs of two different networks can be very different, and often cannot be implemented using the same technology.

For example, for any transmission medium, there is a tradeoff between size and speed: a Local Area Network, linking computers in a single building, typically transmit data between 4Mb/s and 2Gb/s, while a Wide Area Network, which can span a continent, runs between 9,6Kb/s and 45Mb/s. (The technical choices are of course further limited by economic considerations: speed is always expensive).

All these practical facts explain why it was impossible to simply "plug" all the networks together. A new method had to be developed to interconnect heterogeneous hardware together: **internetworking**.

II.2-Some basic (inter)networking principles

II.2.1-Dedicated connection vs. packet switching

There are two basic ways for data to travel on a network: using dedicated connections or using packet switching.

II.2.1.1-Dedicated connections

In this case, the user initiating the communication asks for a connection with another user. If the request is accepted, a physical circuit is opened between both users. They can then begin sending and receiving data on this dedicated line. They will release the connection when the exchange of information is finished.

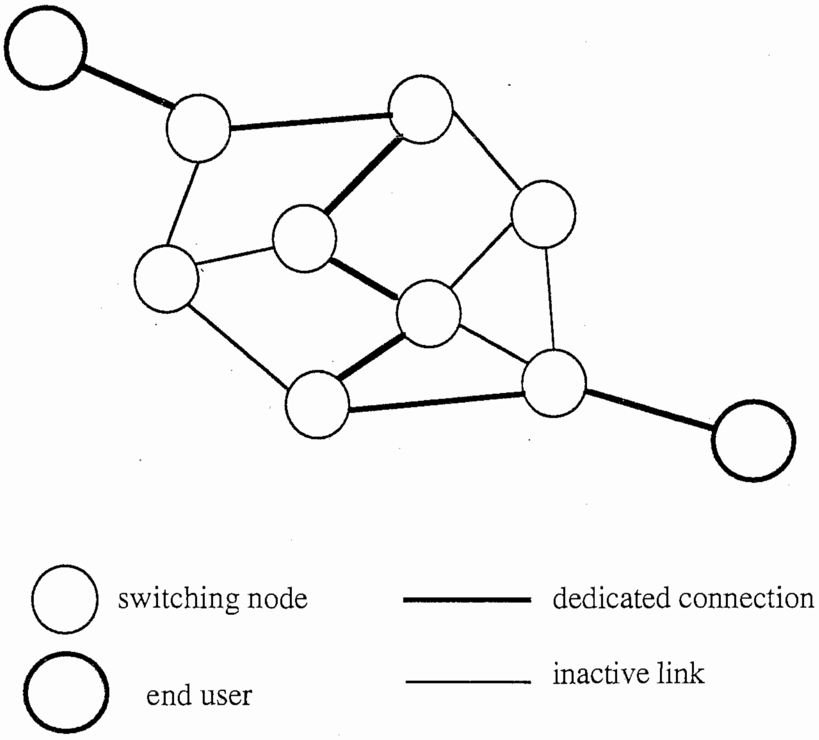

fig 2.1: Dedicated connection

The main advantage of this method is that once the connection is granted, the users can be sure they will have the full capacity of the line reserved for their use. The drawback is of course that nobody else can use the physical line, even if the users are not actually transmitting anything on it.

The best example of a network working with dedicated connections is the telephone network.

II.2.1.2-Packet switching

In a packet switching network, the user's data is divided into pieces of a fixed maximum size. The destination address is then added to each piece, which is now called a **packet**. The network is composed of packet switches. These basically receive a packet, check its destination address, and send it to the neighboring switch which is known to be the next step on the path to the final destination (these routing decisions are made independently for each packet of a same message, which explain why two packets from the same message can follow different paths).

This way many packets from different users can be sent over the same physical link, and the use of the link is optimized. The drawbacks are that links can overload, packets can be delayed or lost, and because different packets from a same communication can follow different paths from source to destination, packets can arrive out of order.

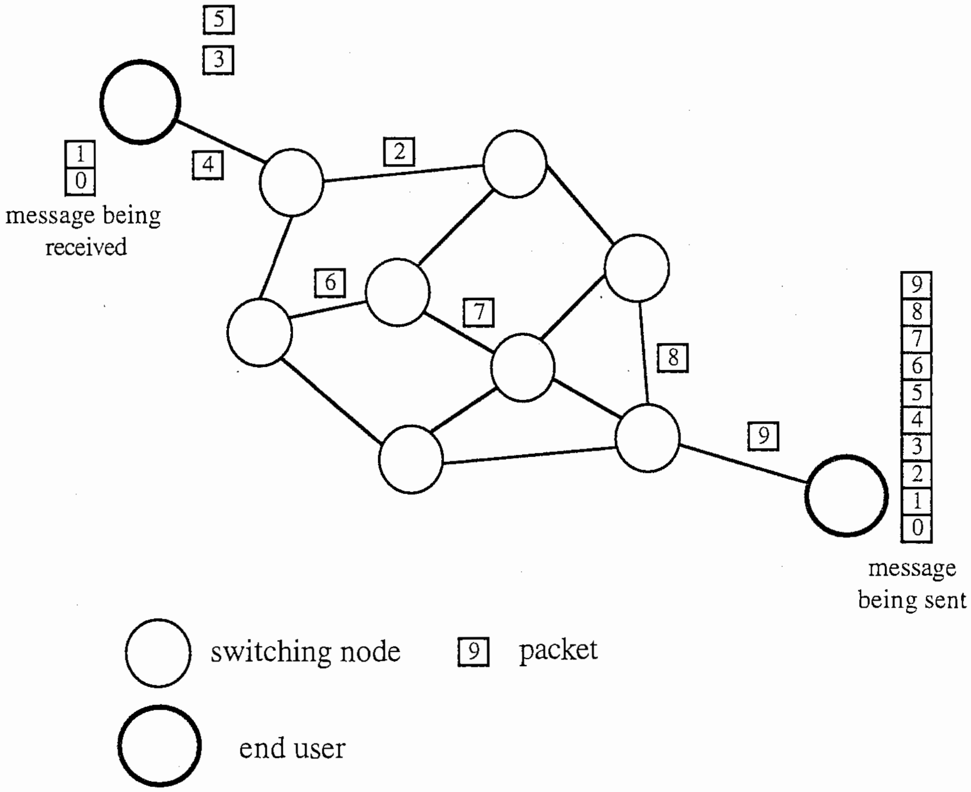

fig 2.2: Packet switching

Despite these drawbacks, packet switching networks are now the norm for computer networks. The Internet itself is based on packet switching technology, as we saw before, and so are its protocols.

II.2.2-Protocol layering

II.2.2.1-Protocols

Firstly, what is a protocol?

A **protocol** is a set of rules determining the way to pass messages. It specifies the format of the messages, and explains how to handle error conditions. A protocol is the grammar that computers have to use in order to communicate with each other.

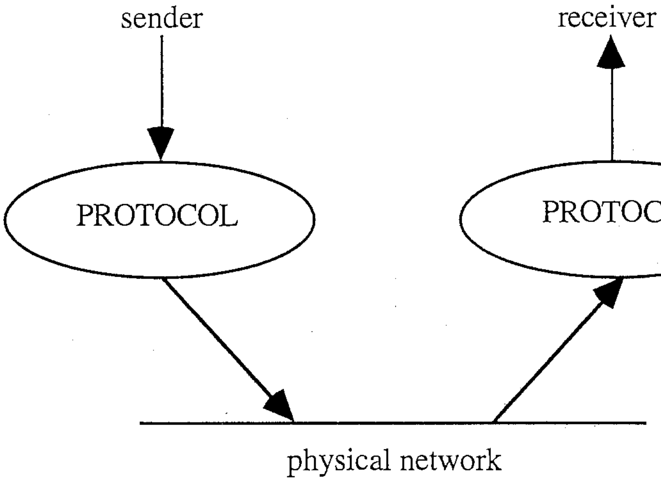

fig 2.3: protocol

Because a protocol is a high level communication language, it is independent from hardware. On the other hand, the software that implement the protocol on a given architecture is hardware dependent. Moreover, since it is also what gives the hardware the ability to "speak" according to a specific protocol's rules, it is also protocol dependent.

This software is called the **interface** between hardware and protocol.

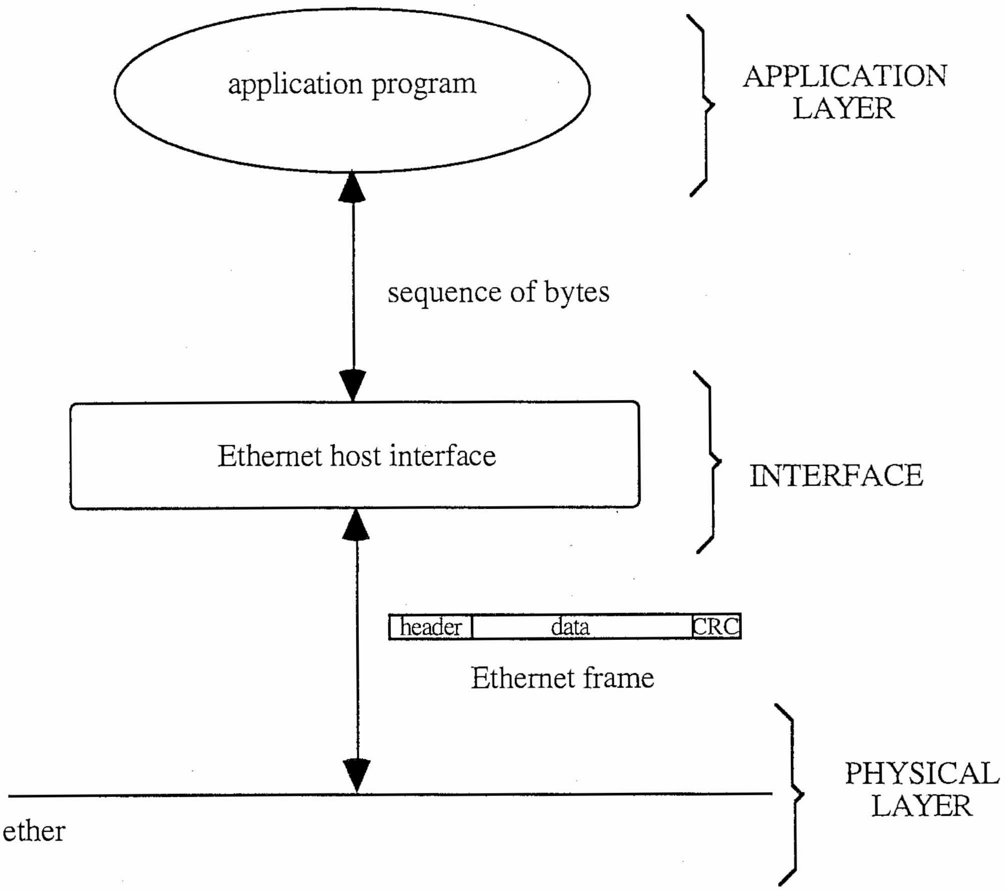

fig 2.4: interface between application program and Ethernet network

What is the interest of protocols, then, if specific code must be written to "teach" the protocol to every single architecture?

We must remember that a network is the medium used by an application program to exchange data with another application program. If no protocol is used, each implementation of each application program has to know how to use the particular network it is running on to be able to work properly.

With a protocol, the interface knows the details of the specific hardware, and the same application program, using only the rules specified the protocol, can be used on every network. The interface simply translates between the universal language of the protocol and the specific dialect of the hardware.

To summarize, let's consider *p* application programs that running on *n* different architectures. Without a protocol we must write *n*p* pieces of code, when we would only need *n+p* with a protocol (n interfaces, p application programs). (Of course, this presents no interest if *n=1*, but then there is no problem of connectivity if all networks are the same).

II.2.2.2-Layering principle

Protocol layering pushes the idea of hiding the particularities of hardware from application programs one step further. A network must be able to do a lot of things to function properly: formatting messages, finding addresses, handling errors...

To handle so many tasks, it is better to use a set of cooperative protocols, called a protocol suite, rather than a huge, monolithic structure. This way, the complex task can be divided between smaller units of protocol software which will be easier to understand and to maintain. These units are called (protocol) **layers**.

Not only does this make writing networking software easier, but it also makes the communication process completely transparent. One layer simply hands the data "up" and "down", and needs only to know about the layers immediately above and under it.

The figure 2.3 can then be replaced by the following one:

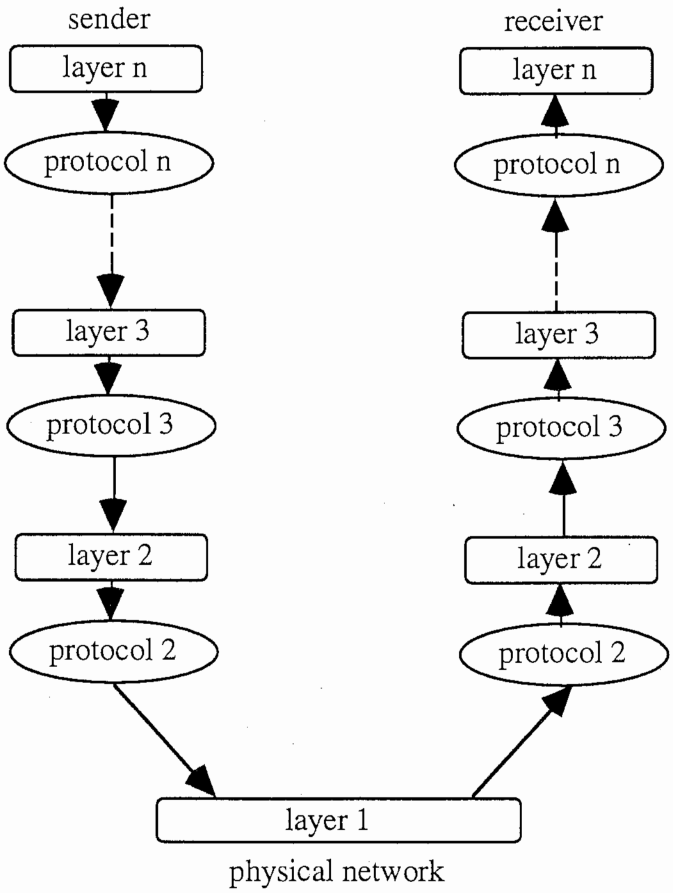

fig 2.5: protocol layering

This principle is called **protocol layering**. Data goes down from the top layer (the application program) to the lowest one (the physical link), through every layer. Each one provides a set of specific functionalities to the communication (error handling, segmentation, acknowledgment...).

As an illustration, here is the International Standards Organization open systems interconnection model for computer communications:

7	application layer
6	presentation layer
5	session layer
4	transport layer
3	network layer
2	data link layer
1	physical layer

fig 2.6: OSI 7-layer model

(Actually, the ISO standard is very complex, and is not yet fully implemented by any existing protocol suite).

The layering principle greatly simplifies network communications because it provides completely transparent communications. Since a layer communicates with its neighboring layer using a given protocol, from its point of view it could be, and is, communicating with the same layer on the other side of the communication. This is illustrated by the next figure, where data follows the solid lines, but seems to be flowing along the dashed lines, from each layer's point of view.

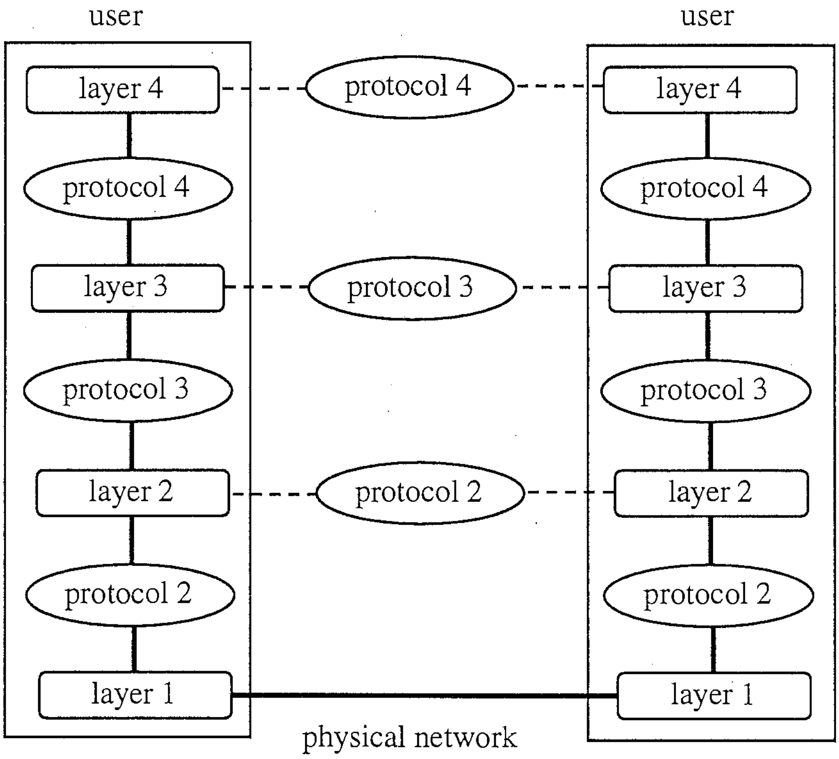

fig 2.7: transparent communications (4-layer model)

II.2.2.3-Encapsulation

One of the tasks done by layer i is to translate data from the format specified by protocol i to protocol i+1. Because data is mostly circulated in packet-like format, this can be done very easily.

A packet usually looks like this:

header
data

where the header contains administrative information for the layer, such as addresses, sequence number...

When layer i receives a packet from "upstairs" (expressed according to the rules of protocol i+1), it reads the header to get basic information (destination address...). It then creates a new header in the grammar of protocol i, and attaches it to the beginning of the packet it received. The result is a protocol i packet, where the data part is the protocol i+1 packet.

When layer i on the other side receives a protocol i packet from "downstairs", it cuts out the header, reads the information it contains, and acts on them. Then it hands the data "up". Since the data part of the protocol i packet is a protocol i+1 packet, it will be understood by the layer i+1.

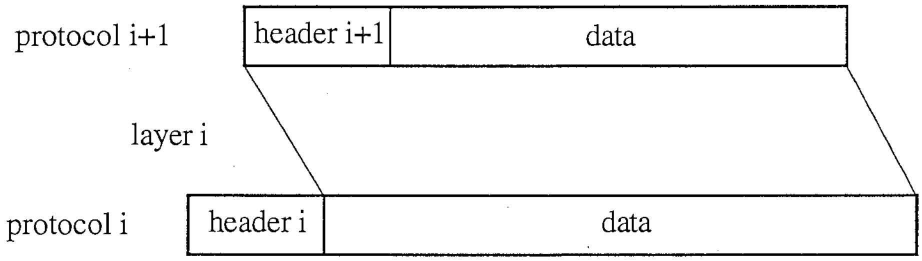

fig 2.8: encapsulation

This method, as shown in figure 2.8, is called **encapsulation**.

Most of the methods and principles we have introduced deal with network communication. We will see in the next part how the architectural model that was chosen for internets makes them easy to transpose to internetwork communications.

II.2.3-Architectural model

We saw that the goal of an internet is to provide universal interconnection (while allowing individuals to use whatever network hardware is best suited to their needs).

First, how is interconnection possible between two networks using heterogeneous architectures?

Two such networks are connected through a special (and usually dedicated) computer called **gateway** or **router**.

This computer is physically connected to both networks. It reads data coming through one of its two network interfaces, and (if the destination of this data is on the other network) it puts it on the other network through its second network interface.

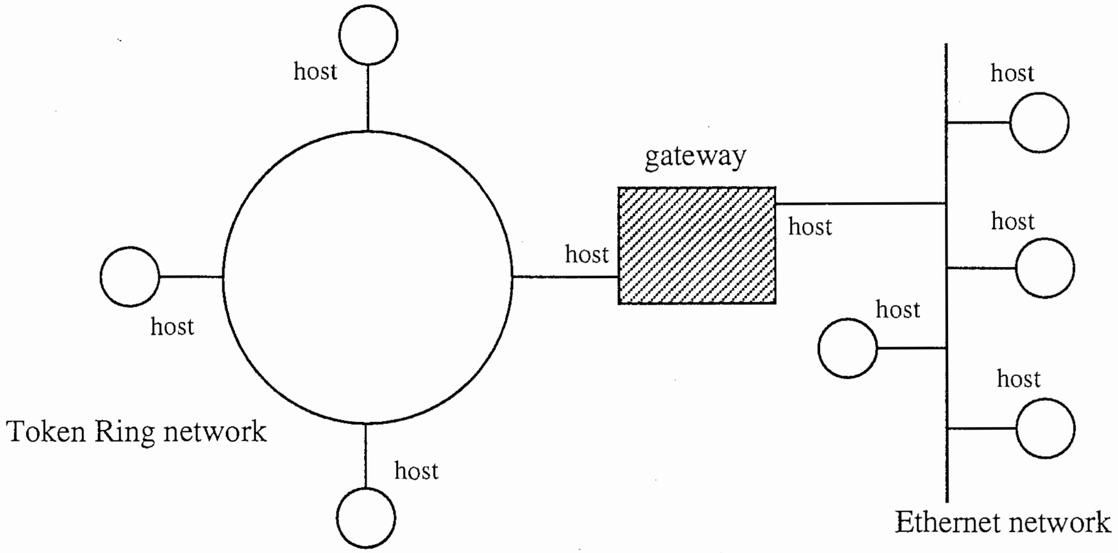

fig 2.9: example of interconnection between an Ethernet and a Token ring network

But such interconnection is not enough to make an internetwork. We also want the topology of the internet and its constituting elements to be free. For example, a new network being connected to an internet should not need to be connected to every other network. This implies that data should be able to cross one (or more) intermediate networks before reaching its ultimate destination.

As a consequence, to be able to route data across the internet, gateways needs to know the topology of the internet. Routing is a complex problem, and an internet makes it even more complex. We will see how it was somewhat simplified by routing data according to network instead of host destination, thus making routing tables much less bulky.

To provide universal service, the Internet uses a scheme that hides the details of underlying network hardware.

Interconnection could be provided at application level, but we saw in the previous section how difficult it is to maintain such a system for a simple network. A complex internetwork, where application programs will have to run on widely different hardware architectures, makes this solution even less attractive.

Instead, interconnection is provided at network level. Data communication activities are then separated from applications, making the system more flexible and easier to adapt to changes, either in network architecture or in application programs. Of course, the work of the application programmer is also made much easier.

Network level-interconnection provides universal interconnection, and makes our heterogeneous collection of networks appear as a single large packet switching network to the user, where the gateways would be the switches (cf. figure 2.10). This is what is properly called an **internet**.

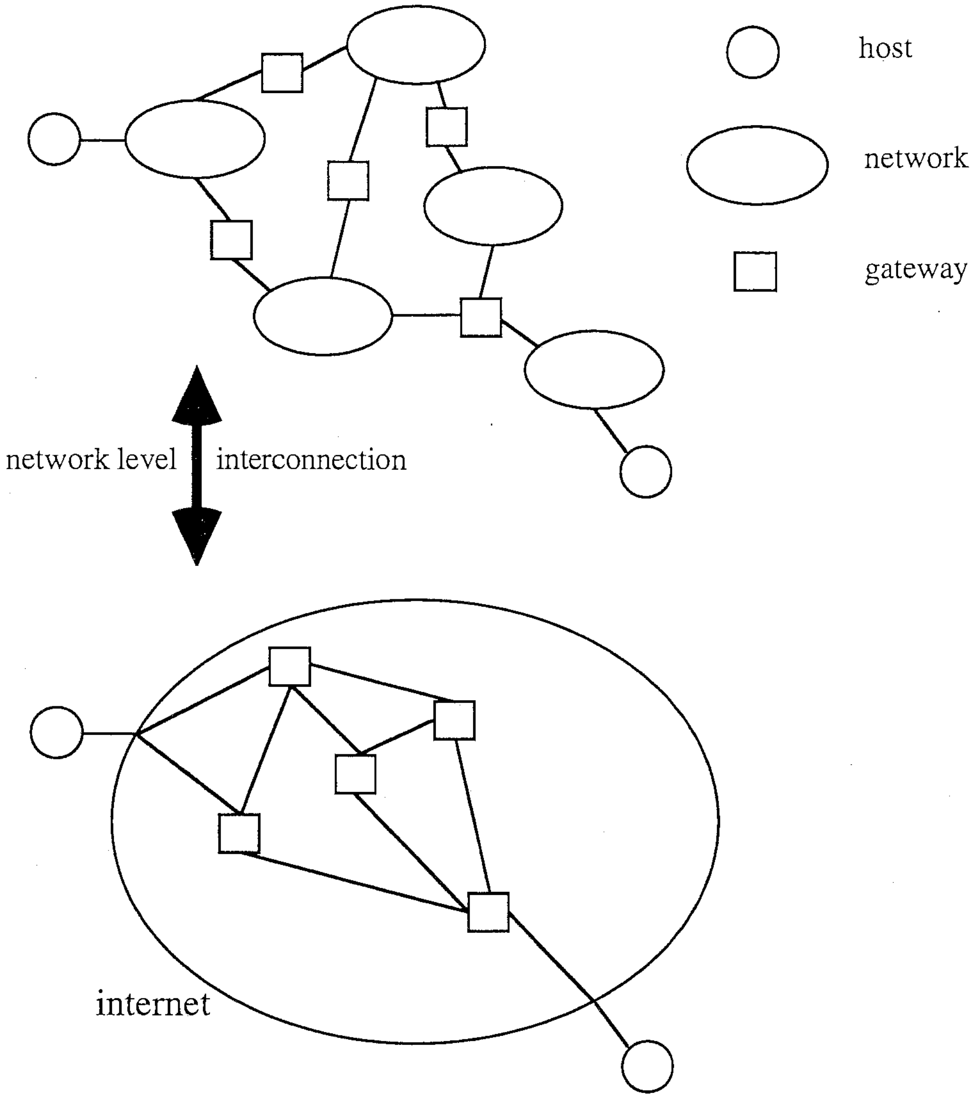

fig 2.10: internet

The use of protocol layering and more precisely of the protocol suite called TCP/IP made this possible.

II.3-The TCP/IP protocol suite

II.3.1-The TCP/IP internet layering model

TCP/IP protocols are built on a simplified five-layer model (cf. figure 2.11) which was adapted to the internetworking concept and architectural model.

The **physical layer** is the hardware connection between hosts on a given network.

The basic unit of communication is the **bit**.

The **network interface layer** puts the network hardware frames together, and map IP addresses to hardware addresses. It handles communications between machines on the same physical network. It is of course hardware specific, and its characteristics and abilities differ widely between different implementations.

The basic unit of communication is the **frame**.

The **internet layer** is the only internet specific layer. It insures communications between machines on different physical networks, thus providing the layers above with a virtual network connecting all the hosts which are actually on the internet.

So the internet layer is the one handling all the complex routing decisions required by an internet.

The service provided is actually an unreliable, best effort, connectionless packet delivery system. Unreliable because packet delivery is not guaranteed. Connectionless because each packet is treated independently from the others. Best effort since unreliability appears only because of exhaustion of resources or failure of underlying networks. As we said before, the internet layer presents the layers above the image of a packet switching network, while dealing underneath with an internetwork.

The basic unit of communication is the **IP datagram**.

The **transport layer** provides end-to-end communication, i.e. communication between two application programs. The characteristics of this layer depends much on the protocol used. The TCP/IP protocol suite proposes two protocols with very different abilities at this level, TCP and UDP.

UDP provides unreliable, best effort, connectionless packet delivery service. TCP provides reliable stream delivery.

The basic unit of communication is the **UDP datagram** or the **TCP stream**.

The **application layer** consists of the application programs used to access services across the Internet (so user's programs are usually one level above the application layer).

The basic unit of communication is a **message** or a **stream**.

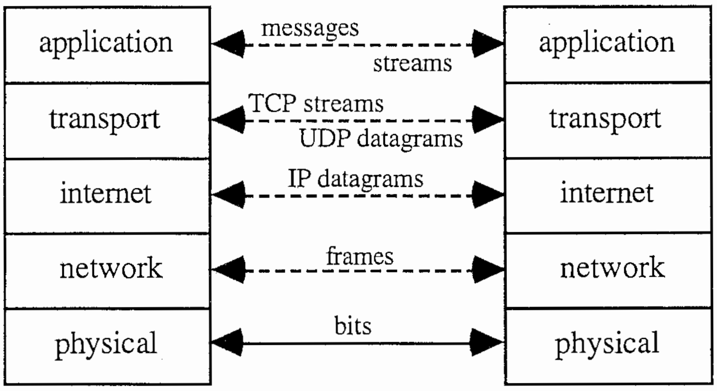

fig 2.11: TCP/IP internet layering model

II.3.2-Internet layer protocol: IP

The IP protocol specifies the various conventions necessary to hide the physical details of an internet. It provides the layer above with an **unreliable connectionless packet delivery service.**

II.3.2.1-Internet addresses

To build the virtual network which will hide the heterogeneity of the internet, the first thing we need is an universal addressing scheme, so that every host on the internet can be uniquely identified.

(Since the virtual network simulated by the IP protocol is implemented in software, designers were free in their choice of an addressing scheme, and could fit it to the other requirements for the IP layer).

IP addresses are encoded as a pair: netid (identifier for a network) / hostid (identifier for a host on a network) in a 32-bit integer. This division was introduced to allow a quick identification of the network involved because, as we mentioned in the previous section, the IP software routes datagrams according to their destination network. (It should be noted that as a consequence, gateways, which are connected to more than one network, have more than one IP address).

There are 5 classes of IP addresses, each one with a different format for the pair netid/hostid. The highest order bits determine the class, as can be seen on figure 2.11 below.

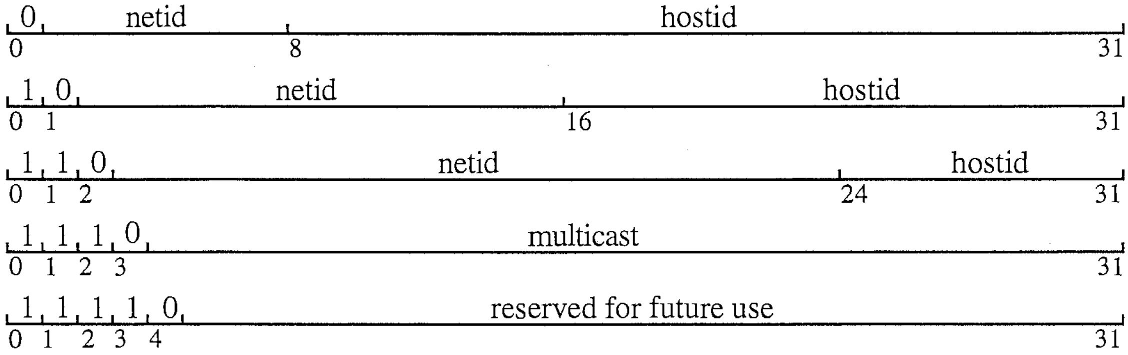

fig 2.11: IP addresses classes

The introduction of different classes of IP addresses was meant to accommodate networks of very different size. Class A addresses, which allows 24 bits for the hostid, was intended for networks with more than 2^{16} (65,636) hosts. Class B networks are intermediate size, counting 256 to 65,535 hosts, and class C networks are small ones (such as Token Ring networks) with less than 256 hosts connected.

Of course, the number of class A network addresses available under this scheme is much smaller than the number of class C networks.

There are two reserved hostids that cannot be used to refer to a real host: netid + all '0' is the IP address for the network, netid + all '1' is the IP address for broadcasting to the network.

Another special address is class A, netid = all '1', hostid = 0, reserved for testing and interprocess communication on a same machine. It simply loops back to its origin, without actually going out on any network.

To make IP addresses easier to read for human beings, they are usually written in dotted decimal notation: four decimal integers, each one representing an octet, separated by dots. For example, using this notation, the loopback address would be 127.0.0.0.

Finally, the Network Information Center insures that netids are unique across the Internet itself. The organization responsible for a network must in turn make sure that hostids are unique on this network.

II.3.2.2-IP datagrams

We saw that the internet layer was providing a connectionless packet delivery service using IP datagrams. To insure such a service, the IP software performs routing and error handling operations. All these tasks are controlled by special fields in the header of the IP datagrams (cf. figure 2.12).

<table>
<tr><td>version</td><td>head. len</td><td>service type</td><td colspan="3">total length</td></tr>
<tr><td colspan="3">identification</td><td>flags</td><td colspan="2">fragment offset</td></tr>
<tr><td colspan="2">time to live</td><td>protocol</td><td colspan="3">header checksum</td></tr>
<tr><td colspan="6">source IP address</td></tr>
<tr><td colspan="6">destination IP address</td></tr>
<tr><td colspan="5">IP options (if any)</td><td>padding</td></tr>
<tr><td colspan="6">DATA</td></tr>
<tr><td colspan="6">...</td></tr>
</table>

fig 2.12: IP datagram

The SERVICE TYPE field allows the user to specify how the datagram should be handled (precedence and type of transport). This gives more information to routing algorithms, and gives them some clues to optimize routing according to hardware capacities. However, the internet protocol does not guaranty that the type of transport required is the one that will actually be used.

The Maximum Transfer Unit (or MTU) of a given network is the limit on the size of the data field for a frame. It is hardware dependent.

Encapsulation requires the size of a datagram to be at most the MTU for the network it is traveling on. Instead of putting a general limit on the size of IP datagrams, they are sent with a size adapted to the first network they cross. If they have to go through a network with a lower MTU, the IP software on the gateway divides the datagram into smaller ones. This is called fragmentation.

When the datagrams reach their final destination, reassembly of the different fragments takes place (if necessary).

The IDENTIFICATION, FLAGS and FRAGMENT OFFSET fields control fragmentation and reassembly.

The TTL (Time To Live) field specifies how long the datagram should be allowed to live. It is decremented by gateways along the way. If it reaches zero before the datagram gets to its destination, the gateway will discard the datagram.

This insures that no datagram will loop forever on the internet, in case routing tables get corrupted.

The PROTOCOL field specifies which higher level protocol (basically UDP or TCP) was used when creating the message in the data field. It tells the IP software at the destination which transport layer protocol should get the data field of the datagram.

The OPTION field is primarily used for network testing and debugging. (For more details, see [10]).

We saw that much of the information in the header of an IP datagram, including of course the source and destination addresses, were used by gateways. The software in gateways has the difficult task of handling routing.

II.3.2.3-Routing

Routing is the process of choosing a path over which to send packets so that they can reach their destination.

Direct routing, when source and destination are on the same physical network, is a simple problem (i.e. entirely handled by the layers under IP, and done without going through a gateway): the datagram is encapsulated in a frame, the IP address is mapped to the hardware address by the network interface software, and the frame is sent to its destination.

Indirect routing is the case when the datagram must cross at least one network to which neither source nor destination host is connected to reach its destination. It is a much more complex problem. The solution consist in dividing it into smaller successive steps.

The originating host knows the IP address of the destination machine, but does not know how to reach it. However, the source knows the address of a gateway on its own network. So it uses direct routing to send the datagram to this gateway. The gateway, upon receiving a datagram, extracts the netid part of the destination address. Using its routing table and routing algorithm, it gets the name of the next gateway on the path to the destination. Since it shares a network with this other gateway, it uses direct routing to send the datagram to its next destination.

Hopping from gateway to gateway, the datagram finally (hopefully) reaches a gateway connecting to its destination network. This gateway hands the datagram to its network interface on the destination network, where the destination IP address is mapped to the destination hardware address.

It must be noted that, while routing is handled by the IP layer, the source and destination addresses in the datagram header are never modified. The IP software in the gateway simply hands down to the network interface software the datagram and the IP address of the next hop. (The routing tables, maintained at the IP level, do not use hardware addresses, as it would defeat the layering principle and make the IP software hardware dependent).

II.3.2.2-ICMP: error and control messages

IP provides, by definition, an unreliable service. Hardware failure, host disconnection, congestion can lead to failure in delivering datagrams.

The Internet Control message Protocol (ICMP) is a special purpose mechanism which deals with such problems. ICMP is used by gateways to report delivery problems, and by hosts to test whether destinations are reachable (ICMP is the basis for the ping command).

ICMP is an error reporting mechanism. It only reports error conditions to the source. It relies on the source to take the actions necessary to correct the problem. And since IP datagrams do not record the path they followed, as they only carry source and destination address, the source must also locate the problem if it occurred on the way.

ICMP also carries control messages. For example, if a gateway overloads, cannot route packets quickly enough, and begin dropping some of them, it can send a "source quench" message to the senders, demanding that they slow down their transmission.

Gateways also use ICMP to exchange routing information ("redirect" message). And it is ICMP which is used to warn the source that a datagram had to be discarded because its TTL expired.

ICMP messages have different formats, always beginning by an octet identifying the type (and the format) of the rest of the message. They travel encapsulated in an IP datagram with a protocol field set to 1.

This should not lead us to mistake ICMP for a higher level protocol: the information in the ICMP messages are formatted, sent, received and processed at the IP layer. The specifications for ICMP are part of the IP standard.

The previous sections reviewed the workings of IP, and the mechanisms that allows it to provide an unreliable, best effort, connectionless packet delivery service to the layers above. The next ones will study these layers above.

II.3.3-Transport layer protocols: TCP and UDP

We saw that the TCP/IP protocol suite provided two protocols with very different capabilities, TCP and UDP.

Messages coming from the transport layer are handed upward by the IP software to one of these two protocols, according to the value in the PROTOCOL field of the IP datagram the message arrived in.

II.3.3.1-User Datagram Protocol

UDP provides, like IP, an unreliable, connectionless, packet delivery service. This means that the layers above must insure reliability itself. This is specially important on an unreliable internetwork such as the Internet.

But UDP still adds a functionality to IP. A transport protocol must insure end-to-end communication, i.e. communication between processes. So UDP provides a mechanism to distinguish between multiple destinations on a given host.

Identifying the ultimate destination is not as easy as it seems. Processes on a machine are known to the operating system through an id number. But these process id change dynamically as processes are started or killed, and even more if the machine is rebooted. It is impossible for every machine to keep up to date with the PID associated at any time with every application program on every other host on the network.

So instead of PID, UDP uses a set of abstract destination points called **protocol ports** to identify processes. The operating system on every machine is then expected to keep track of the relationship between protocol port and PID, and to provide an interface between processes and ports. The access to incoming data is synchronous and buffered.

An UDP port can be thought of as a queue holding incoming messages

Protocol ports are 16-bit unsigned integers. A UDP datagram carries both source and destination protocol ports, to make answering to a message easier.

The format of a UDP datagram is shown in figure 2.13.

UDP source port	UDP destination port
UDP checksum	UDP message length
DATA	
...	

fig 2.13: UDP datagram

As we can see, the source and destination IP addresses are not included in the UDP datagram. This means that the UDP software must interact with the IP software to get them (they are needed to send the answer). In theory, the IP and UDP layers can only exchange UDP datagrams. Obviously, the TCP/IP protocol suite violates the protocol layering principle in this case. This violation is allowed because it is compensated by a greater efficiency.

The last question raised by the UDP protocol is: how are protocol ports used? How does an application program on a specific machine knows which protocol port is associated to the application program it wants to communicate with on another machine?

Some port numbers, called "well-known" port numbers, are assigned by a central authority. They are used by the most common application programs (e.g. echo = 7, daytime = 13, trivial file transfer = 69...).

The majority of other port numbers are assigned dynamically by each machine to (less common) application programs. To learn the protocol port associated with such an application on a particular machine, the sender first asks the protocol port associated with the name of the application on the destination machine, addressing the question to a special well-known protocol port on the destination machine. This special port number is itself associated with a piece of software called a portmapper, which keeps track of the association between dynamically assigned port numbers and application programs on its own host.

II.3.3.2-Transmission Control Protocol

TCP defines the second most important internet service (after connectionless packet delivery): **reliable stream delivery**. TCP is a transport layer protocol, and, contrary to UDP, adds subsequent functionalities to the service provided by the internet layer.

As a consequence, the TCP protocol is much more complex than UDP. Because of this, and because the present version of Tierra only uses UDP datagrams for communication, we will only make a short presentation of the TCP functionalities in this chapter.

Why is reliable stream delivery such an important service?

Application programs often need to send large volumes of data, and also need the transmission to be reliable. TCP was introduced to allow application programmers not to worry about error detection and recovery. The TCP software take care of this, and provides the application program with a reliable transmission.

Reliability is insured through positive acknowledgement and retransmission: the destination machine confirms the arrival of every piece of information sent by the source, using a special message called an acknowledgement (ACK message). If the source does not get this acknowledgement, it sends the data segment once more. To avoid duplicate segments if it is the ACK that gets lost, segments are given a sequence number, and if the destination receives a retransmitted segment it already got, it discards it.

The interface between and application programs has other useful properties:

Stream orientation: the two user processes communicating through TCP exchange streams of bytes. They do not have to divide their messages into packets, and then reassemble them on destination: the TCP software takes care of this.

Virtual circuit connection: actually, the TCP software provides its users with the illusion of a dedicated hardware circuit. The users get the advantages of a connection oriented network, while the use of hardware equipment is optimized by the underlying packet switching network provided by IP.

Buffered transfer: to make transfer more efficient, the data handed to the TCP layer by the application program is buffered before being actually transmitted.

Unstructured stream: application programs using TCP must agree on stream format before initiating transmission, using a three-way handshake.

Full-duplex connection: TCP allows concurrent transfer in both directions on the virtual connection.

As we saw, TCP software divides the data stream it receives from the application layer into segment (cf. figure 2.14).

<table>
<tr><td colspan="3">source port</td><td colspan="2">destination port</td></tr>
<tr><td colspan="5">sequence number</td></tr>
<tr><td colspan="5">acknowledgement number</td></tr>
<tr><td>head. len</td><td>reserved</td><td>code bits</td><td colspan="2">window</td></tr>
<tr><td colspan="3">checksum</td><td colspan="2">urgent pointer</td></tr>
<tr><td colspan="4">options (if any)</td><td>padding</td></tr>
<tr><td colspan="5">DATA</td></tr>
<tr><td colspan="5">...</td></tr>
</table>

fig 2.14: TCP segment

The many fields in the header show that the workings of TCP are actually much more complicated than the brief outline we gave. TCP allows out-of-band data (treated as soon as it arrives, without waiting for its turn in the incoming queue), and many options to provide maximum efficiency and reliability. For more information on these subjects, see [10].

Like UDP, TCP uses protocol ports numbers to identify the ultimate destination of a segment. But TCP uses the **connection**, i.e. a pair of endpoints of the form {destination host/port, source host/port}, as its fundamental abstraction.

TCP uses the same mix of static (255 well-known port numbers maximum) and dynamic port bindings to link application program to port number. Because UDP datagrams and TCP segments are differentiated at the IP level, the port numbers for each protocol are independent. Still, the designers chose to assign the same port numbers for services that are available through both TCP and UDP.

II.3.4-Interface to the transport layer protocols: sockets

To be able to use the TCP/IP protocol suite, application programmers need an interface to the transport layer. However, such interfaces are operating system-dependent, and so there is no standard for them.

This section will briefly introduce the socket interface, which is the one used by 4.3BSD Unix OS. Its many useful and efficient features have been widely copied, but it still is not a standard. But first we will study the client-server model of interaction, which provides the basis for most network communications.

II.3.4.1-Client server model of interaction

A **server** is a program offering a service that can be reached by sending a request over the network. It is usually an application level program, which waits for a request to arrive. When it receives a request, it performs some actions based on it, often including sending an answer.

A **client** is any process that send a request to a server.

Servers can be divided into two categories. **Iterative** servers, upon receiving a request, perform the associated action(s). A **concurrent** server waits for a request, and, when receiving one, creates a slave process. This slave handles the actions associated with the request, while the server process resumes its waiting. This insures that the server will not get tied with one request, and ignore incoming messages.

The figures in Annex 1 shows how the client-server model is implemented using sockets for communication.

II.3.4.2-Sockets

The **socket** is the basis for 4.3BSD Unix network I/O. It is an endpoint for communication. The designers of the socket tried to have it behave just like the Unix files and devices (whenever it made sense).

Sockets are created, maintained, used and destroyed using a set of special system calls.

A socket is first created (`socket` system call). It is associated to a protocol (TCP or UDP for the TCP/IP protocol suite) at birth. The user receives a socket descriptor, a small integer which will be the identification number for all further access to the socket.

The `bind` system call is used to associate a local protocol port number to the socket. The socket can now receive data. If it is to be used with stream service, it must also be bound to a destination address. This is done through the `connect` system call.

The `connect` system call can be applied to a socket using a packet delivery service, but then it simply provides automatic addressing for outgoing packets. When used with a stream service, it also takes care of all the details associated with the opening of a virtual connection.

Data can now be sent using the socket. This is done using `write`, `writev`, `send` (connected socket only), `sendto`, `sendmsg`.

To receive data, `read`, `readv`, `recv` (connected socket only), `recvfrom`, or `rcvmsg` should be called.

The `listen` system call is used to assign a queue length to a connected socket.

The `accept` system call tells a TCP socket to wait for a connection. It is blocking, and returns a new socket descriptor when accepting the connection.

The `select` system call allows a single process to wait for a connection on multiple sockets.

The socket interface also provides a number of library functions to handle data transmission, and to get addresses, protocol numbers and names, as well as information about networks and hosts. The details of all these functions can be found in [11].

This chapter exposed the basic workings of the Internet and its protocols. These will only be accessed through the socket interface, but knowing how a network switches packets around, maintains routing information and basic data about its hosts, and handles error conditions will prove important to the conception of the virtual network Tierra presents to its creatures.

III-Tierra, network version

We have seen what Tierra is, why we need a network version of Tierra, and how the Internet is working.

The only (!) question left is how to put Tierra on the Internet.

When I arrived in ATR, a beta version of a networked Tierra for an Unix environment was already halfway implemented. This chapter first explains what technical choices were made for this beta version, and how it was working. A description of the work I did follows, leading to a first version of Tierra (dubbed network 1.0 in the following text).

Last, but not least, are a few of the features that should be implemented in a future, more evolved, second version of Tierra.

Note:

A few things had to be kept firmly in mind during the conception of the network version of Tierra. The first one is that Tierra must be as unobtrusive as possible. As a consequence, the Unix version must run with the lowest possible priority.

We also want anybody to be able to run its own Tierra. For an Unix standard system, the consequence is that none of the functionalities used by Tierra should require root privileges to run.

III.1-Network Tierra, β version

III.1.1-Getting there

III.1.1.1-Non-network Tierra

Tierra is a C program implementing a virtual computer. So the C program must read the virtual code (i.e. the genome of the creatures), execute it, and take care of all the administrative details an operating system usually deals with (such as memory allocation...).

Here is the outline of the `main()` of Tierra (as found in tierra.c).

```
int main(argc,argv)
int argc;
char *argv[];
{
  GetSoup(argc,argv);
  life();
  WriteSoup(1);
}
```

`GetSoup()` (in tsetup.c) initializes the soup, which is the segment of memory where the "pseudo-code" of the creatures will be read and written. The language used by Tierran creatures is a simple assembler language with a set of 32 instructions. The actual instruction set (i.e. the array mapping between assembler mnemonics, 5-bit opcodes, and

C functions executing the instructions on the real machine) is part of the information gotten by GetSoup().

WriteSoup() saves the soup to disk.

life() runs in a loop for a maximum of alive generations. It doles out the time slices (Tierra simulates a parallel machine through time slicing, executing part of the code of each creature in turn). And it calls ReapCheck() to kill a (some) creature(s) if there is not enough memory free for a new creature to be born.

```
void life() /* doles out time slices and death */
{
  while(Generations < alive) {
    (*slicer)();
    ReapCheck();
  }
}
```

Tierran creatures are the programs running on the virtual computer. They are represented by their code, stored as a sequence of bytes in the soup, where one byte encodes one instruction. Tierra also maintains a structure (in the file tierra.h) containing administrative details about the creature (pointer to the code in the soup, value of the instruction pointer for the creature, content of the registers...).

```
struct cell {   /* structure for cell of organisms */
  Dem d; /* fecundity, times and dates of birth and death */
  Que q; /* pointers to previous and next cells in queues */
  Mem mm; /* main cell memory */
  Mem md; /* daughter cell memory */
  CpuA c; /* virtual cpus */
  I8s ld; /* 0 = dead, 1 = alive */
};
```

This cell structure contains the demographic data Tierra gathers about every creature, pointers to the memory of the cell, and a pointer to an array of CPU.

```
typedef struct { /* structure for cpu array */
  I16s ib;     /* instruction bank */
  I32s n;      /* number of allocated cpus */
  I32s ac;     /* number of this active cpu */
  I8s  sync;   /* sync flag for this cell */
  Cpu  *c;     /* pointer to currently active cpu */
  Cpu  *ar;    /* pointer to array of cpus */
  InstDef *d;  /* pointer to current InstDef structure for
                  parsing */
} CpuA;
```

The CpuA structure contains all the data used to manage the array of CPUs for a cell. A creature can have in some cases more than one CPU (but only one can be active at a given time).

```
typedef struct { /* structure for virtual cpu */
  Reg re[ALOC_REG];    /* array of registers */
  Reg ip;              /* instruction pointer */
  Reg sp;              /* stack pointer */
  Reg st[STACK_SIZE]; /* stack */
  Flags fl;           /* flags */
  CRflags cf;         /* CPU Register flags */
  I8s sync;           /* wait for sync */
#if PLOIDY > 1
  I8s ex; /* track of execution */
  I8s so; /* source trace for reads */
  I8s de; /* destination track for writes */
  I8s wc; /* wait count for error-track switching */
#endif /* PLOIDY > 1 */
} Cpu;
```

This Cpu structure is the heart of the creature. Each creature is a small computer, with a complete CPU executing the assembler code stored in its genome. All the registers necessary to run the code can be found in this structure, which is used by Tierra to make the link between the pseudo-code of the genome and the actual C functions run by Tierra to simulate the virtual computer, every time the creature gets a time slice.

III.1.1.2-Moving to the network version

a) The needs

Tierra has to provide its creatures with new abilities and add new capacities to the existing software, in order to allow them to use the possibilities introduced by a network.

The first one is the ability to migrate from one computer to another. This ability to move from node to node could be implemented by allowing a creature to spawn a daughter on a remote node. (This daughter being genetically similar to its mother, this is truly a form of migration.)

The need for another new ability is born from the previous one. If creatures are to be able to migrate to another node, they have to learn how to do this wisely. In short, creatures should look for a better node to move into (better meaning, from the creature's point of view, with more space and energy (CPU cycles) available).

In order to allow creatures to really choose their destination in a migration, instead of selecting a remote node at random, we have to provide them with "feelers", i.e. a way to observe the outside world.

These basic requirements from Tierran creatures led to some initial technical choices for their implementation.

b) Basic choices

The first of these choices was for the communication medium. Since the first version of a network Tierra was to be implemented with Unix as an operating system, **sockets** were the obvious answer. As we saw (in section II.3.4.2), they provide a simple interface between application programs and the TCP/IP protocols through the Unix kernel.

The next step was to choose the protocol these sockets would be using. We saw (in section II.3.4) that TCP provides reliable communications to application programs.

However, there is a big overhead in time implied by the establishment of a virtual circuit between source and destination. Tierra would have to remain "frozen" and as a consequence would be unavailable to run the soup during that time. Furthermore, the messages exchanged between nodes will be relatively small (at most the size of the biggest creature existing in the soup).

These facts plead for UDP datagrams.

The main drawback of datagrams is the lack of reliability. But some reliability can be added inside Tierra itself: we saw that the TCP/IP datagram protocols provide a mechanism to report errors, and that reacting to the errors was the sole responsibility of the layers above.

Beside, we are not really interested in "perfect" reliability (if this ever existed), because we want Tierra to provide to the creatures a simulation of a working network (however simplified). And all networks are to a certain extent unreliable.

Furthermore, Tierra is, by design, an unreliable computer.

So the final decision was to use sockets with **UDP** as a communication protocol.

These choices led to the writing of a first, beta version of a networked Tierra. It only had some primitive communication abilities. But it was able to answer to the basic needs expressed in section III.1.2.1.

III.1.2-Conception

III.1.2.1-General organization

a) Outline of the communication mechanism

As can be seen in figure 3.1, the first version of Tierra's communication system used one UDP socket, managed by Tierra itself.

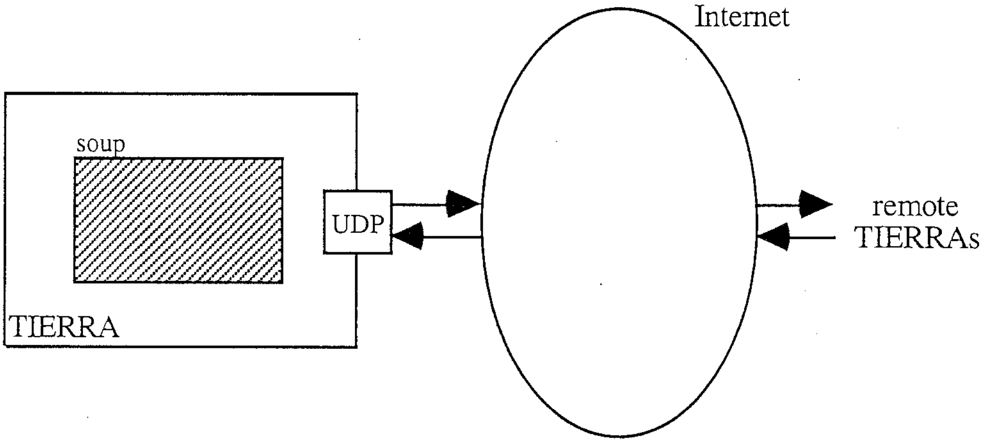

fig 3.1: Tierra β version

Tierra cannot use the usual client-server model: a node must be able both to send requests and to reply to other nodes. Furthermore, Tierra cannot get stuck waiting for messages, because it would then stop the operation of the virtual computer for the creatures.

The simplest solution here was to stick together the outline of both client and server. This way, Tierra functions like a server (obviously an iterative one) a small part of the time, checking for incoming messages on one socket. This is called a **polling** server.

The rest of the time is spent in the "non-network" activities. Tierra runs the infinite loop for the creatures in the soup, and takes care of the necessary administrative tasks. The only difference is that, when the need arises, it will send messages to a remote node, acting in this case as a client to the remote node's server.

These messages are sent at two kind of occasions. Firstly when Tierra executes a part of a creature's pseudo-code that "sends" a message (on the virtual network seen by the creatures). In this case, the creature's "message" must be made by Tierra into something the network code will understand.

Secondly when the local node needs to contact a remote Tierra. As we will later see, some administrative activity is necessary to keep the loosely "connected" network of Tierran nodes coherent. This is handled by Tierra, and is kept well separate of the running of the creatures.

To send a message, Tierra first has to put together the data to be sent in the datagram. The specific socket system calls (shown in section II.3.4.2) are then used to fill in the destination address structures, and to send the datagram.

In the same way, when a message arrives on a socket, the data first has to be decoded. The content of the message is then processed by Tierra, calling an adequate function. If necessary, some information is handed to a creature.

b) Introduction of the "mapfile"

One of the first problems to arise was the management of addresses. Obviously, to send a message to a remote node, the local Tierra must know some basic information about how to reach this node. This is the IP address, and the port number for the remote Tierran process.

A simple solution was adopted. Tierra uses a "mapfile". This is a sequential file containing the addressing information for every site known to the local process.

This information is stored on one line in the following format, and this for every node:

```
name_of_machine:IP_address:port_nb;
```

where name_of_machine is the name information given by the Internet Domain Name Server about the machine (stored as a string). The IP address is stored in dotted decimal format. The port number is an unsigned 16-bit integer (formatted as part of a string).

This information is meant both for Tierra itself and for the user, which can check which sites are (potentially) communicating with the local node. This is the reason why the name (which is completely useless to Tierra) is stored in the mapfile, and why IP addresses are stored in dotted decimal format.

III.1.2.2-Message format

The second problem was about the messages themselves.
First, what kind of messages are needed?

We saw in section III.1.2.1 that Tierra had to provide a way to send a genome to another site, and a mechanism for creatures to gather information about remote sites. Furthermore, the previous section introduced the problem of addresses. To maintain the mapfiles on the different sites, we need a mechanism enabling a node to learn addresses from other nodes.

So Tierra needs at least three sets of messages: one for migrations, one for getting information (dubbed tping, after the Unix ping command that allows a user to know whether a site is alive and responding), and one for address exchanges.

Tierra must then be able to distinguish between the different types of messages being received. A standard format for messages has to be introduced.

A message begins with a tag (encoded on a 32-bit integer) that identifies its type. This tag is followed by the content of the message itself, encoded as an array of bytes.

This general outline is shown in figure 3.2.

tag	data

fig. 3.2: General format for Tierra messages

There are four types of messages. Two types for the tping mechanism, which include requests for information, and replies. One for the migration mechanism. One for sending addresses.

This makes for four different formats, as shown in figure 3.3.

EMIGRATETAG	genome (as an array of bytes)

TPINGREQTAG	source address (creature)

TPINGREPTAG	source address (node)	destination address (creature)	tping data

SENDADDRTAG	sender address (mapfile format)

fig. 3.3: Detailed format for Tierra messages

In these messages, two different types of addresses appear, one for nodes and the other for creatures. The address for a node contains the addressing information necessary to get to a remote Tierra, i.e. an IP address (32-bit unsigned integer) and a port number (16-bit unsigned integer).

The address for a creature is a bit more complex. In order to reach a cell on a remote node we must know the address of the node (see paragraph above), and the address of the creature in the soup.

Let's review these messages in detail.

For an emigration message, the only data necessary is the genome of the creature. Source and destination IP addresses are given directly to the socket system calls, which handle the inclusion of IP addresses and UDP ports into the appropriate datagram headers. So they are not needed as a part of the data.

For a tping request, the destination address is, once again, not useful, because the request is sent to Tierra as a server, through the adequate system calls. But a source address is necessary, because the remote Tierra will have to know where to sent the answer. This answer should reach the creature which asked for the information. And the address of a creature cannot be handled by the underlying layers, because it only depends on the local Tierra.

A tping reply includes the destination address for the cell, and the source address from the node. Tierra could get this information directly from the socket system calls, but we tried to be consistent with the layering principle in this occasion. With a tping reply, a creature is directly interested by the source address (especially if it sent a lot of tping requests, because UDP does not guarantee that the replies will arrive in the same order as the requests).

The tping reply also includes the data requested. This data consists of some environment variables that characterize the way the remote Tierra is running. They are: SoupSize (the size of the soup in bytes), NumCells (the number of cells currently in the soup), and Speed (actually a inverse of a speed, since it is the time it took for the last one million pseudo-instructions to execute). These are standard environment variables for Tierra.

The message used to teach an address to a remote node simply includes the relevant information in a mapfile format, as a string of characters. This is the most convenient method, both for source and destination.

The first part of section III.1 exposed the needs of Tierra, the second one proposed solutions. The next one will show how these solutions were implemented... and the problems that appeared during the implementation.

III.1.3-Implementation

The implementation required a few data structures to be modified or added, a lot of new functions to be written... and all this new code to be inserted in the proper places in the voluminous files that make up the Tierra project.

Note: All the network related code is under `#ifdef NET`, except for the file netfunc.c, which is specific to the network version. This way the network version can be evolved (and released) along with the non network-related code.

III.1.3.1-Data structures

a) ComAddr and ComData

A message is always made up of some basic elements. The following structures represent the address of a creature, and the data field of a datagram.

```
typedef struct {
     I32s ranID;
     I32s cellID;
     I32s soup;
     I16u portnb;
     I32u node;
} ComAddr;
```

This address structure is used to identify uniquely one creature in the network of Tierran nodes. A creature is known through its IP address and protocol port number (node and portnb), its soup (in case Tierra runs more than one soup, as would be the case on a parallel machine), and its cellID, which is actually the address of the creature in the soup.

A ranID field had to be added to locate a creature in time.

Communications, especially over such a slow network as the Internet can be, can introduce very large delays. It is then possible for a message meant for a creature to arrive long after the recipient is dead. If a creature is identified only by its address in the soup, this message will be received by the new "inhabitant" of this particular memory space. The ranID value, which is chosen at random for every new-born creature, allows Tierra to detect such problems.

```
typedef struct { /* structure for communications data */
  I32s len; /* length of message */
  I8s  *d;  /* array for data communications */
} ComData;
```

The data field of a message will be stored in such a structure (dynamic array).

b) IOs and IOb

Tierran creatures communicate to gather information about their surroundings. This information (such as the data sent as a reply to a tping request) must somehow be handed to the creature. The data is stored in the soup, and the IOS and IOb structures where created to allow Tierra to manage these data spaces (for example to know where to put the data previously requested by a specific creature).

```
typedef struct {   /* structure for IO for communication */
  ComData d; /* data */
  ComAddr s; /* source address */
  ComAddr e; /* destination address */
  I32s    t; /* tag for type of message */
} IOS;

typedef struct { /* IO structure (communications buffer) */
  I32s ipi; /* current index into mapfile */
  I16s siz; /* currently allocated size of IOS buffer */
  I16s nio; /* index of next free IOS for incoming message*/
  IOS *io;  /* pointer to IOS buffer; 0 struct for */
} IOb;      /* outgoing, 1 to (siz - 1) for incoming */
```

The ipi field was introduced to allow a creature to search for addresses in the mapfile. This way, instead of playing around with real IP addresses, creatures can only refer to an address in the mapfile.

As an advantage, they do not have to specify a 32-bit IP address plus a 16-bit port number, but only the 32-bit index in the file.

Furthermore, as can be seen above and under, each creature has its own pointer. This allows every creature to go through the whole file at its own pace, thus gathering more or less comprehensive information about the different nodes.

c) CpuA

```
typedef struct { /* structure for cpu array */
  I16s ib;     /* instruction bank */
  I32s n;      /* number of allocated cpus */
  I32s ac;     /* number of this active cpu */
  I8s  sync;   /* sync flag for this cell */
  Cpu  *c;     /* pointer to currently active cpu */
  Cpu  *ar;    /* pointer to array of cpus */
  InstDef *d;  /* pointer to current InstDef structure for
                  parsing */
#ifdef NET
    IOb io;    /* IO buffer for network communications */
#endif /* NET */
} CpuA;
```

The necessary pointer to the I/O buffer was added to the CpuA structure.

III.1.3.2-Different functions involved

Tierra uses two sets of functions, one to send datagram, one to receive and process them. These sets are of course interdependent.

This section first deals with the functions sending messages, then with the receiving function, and last with the functions processing messages previously received.

a) Sending a genome: NEject()

NEject() is called whenever a creature wants to emigrate. For this beta version of Tierra, it is called at random, with a low probability, every time a creature attempts to spawn a daughter.

Here is the prototype of the function:

```
void NEject(I32s gen,I32s siz);
```

The parameters for this function are gen, the address of the genome in the soup, and siz, the size of the genome.

This function copies the genome in a message, and sends it to a remote node. In this beta version of Tierra, the destination address is chosen at random in the mapfile.

```
Create buffer
Put EMIGRATETAG in buffer
Put genome (siz octets from the gen address onward) in buffer
Choose index in mapfile at random
Open mapfile
Get index-th line in mapfile
Extract address information
Fill in destination address for socket with data above
Send buffer contents on socket
Close mapfile
```

b) Sending an address: Addr_send()

`Addr_send()` is called once at startup, and is used by Tierra to broadcast to other nodes its ability to communicate and its address.

```
I32s Addr_send()
```

The return value gives some status information.

This function sends the local address (association `{`IP address; protocol port`}`) and name, in the same format as a mapfile line, to every known node.

It goes through the following loop until it has gone through the whole mapfile:

```
Create buffer
Put SENDADDRTAG in buffer
Put local name and address under mapfile format in buffer
Open mapfile (for reading)
While not end of mapfile
    Get next line in the map file
    Extract address information (IP address and protocol port)
    Fill in destination address for socket with data above
    Send buffer contents on socket
Close mapfile
```

c) Sending a tping request: TPing_send()

`TPing_send()` is called by the decode function associated to a specific instruction in the assembler code of the creatures (`tpingsnd()` in instruct.c). It sends a tping request to a node whose index in the mapfile is specified by the creature.

```
I32s TPing_send(I32u index,ComAddr *cell_addr);
```

The parameters are `index` (index for an IP address in the mapfile) and `cell_addr`, a `ComAddr` structure containing the address of the creature sending the tping. The return value gives some status information.

The outline of the code is the following:

```
Create buffer
Put TPINGREQTAG in buffer
Put cell_addr in buffer
Open mapfile
Get index-th line in mapfile
Extract destination address (IP address and port number)
Fill in destination address for socket with above data
Send buffer content
Close mapfile
```

d) Receiving messages: NetRecvFunc()

This function is called at every turn of the `life()` loop (cf. III.1.1). It checks whether there is an incoming message waiting on the socket. If there is one, it processes the message and returns. If there is no message waiting, `NetRecvFunc()` returns without blocking.

```
I16s NetRecvFunc();
```

The return value indicates the return status of the function.

The outline of the code follows:

```
Create buffer
Read on socket
If no incoming datagram        (this is possible because the socket)
    Return                     (option was set to non-blocking)
Else
    Put incoming data in buffer
    Check message tag
    Case tag is EMIGRATETAG
        Call Inject()
    Case tag is SENDADDRTAG
        Call Addr_recv()
    Case tag is TPINGREQTAG
        Call TPing_reply()
    Case tag is TPINGREPTAG
        Call TPing_recv()
```

e) Processing a genome: Inject()

This function is called from `NetRecvFunc()` when a message carrying a genome is received. It puts the received genome in the local soup.

This function is used to inject in the soup genomes from varied origins (network, genebanker, other soup...). This explains the apparition of some of the parameters.

```
void Inject(g, size, sad, tol, disk, rrpi)
FpInst  g;      /* pointer to genome */
I32s    size;   /* size of genome */
I32s    sad;    /* suggested address for placement */
```

```
I32s    tol;    /* tolerance placement of genome */
I32s    disk;   /* 1 = this genome comes from the disk */
float   *rrpi;  /* reap rand prop for injection */
```

This function receives the address and size of the incoming genome (g and size parameters), which is presently located in a receiving buffer.

An address in the soup can be suggested for grafting the new genome in the soup, through the sad and tol parameters.

f) Processing an address: Addr_recv()

This function is called from NetRecvFunc() when an address message arrives. It copies the received address in the local mapfile, if this address was not already known.

```
I32s Addr_recv(I8s *address);
```

This function receives as parameter the string (in mapfile format) received in the datagram. It returns an exit status.

The outline of the code for this function follows:

```
Extract IP address and port number from string
Open mapfile
While not end of mapfile
    Get next line in the mapfile
    Extract address information
    Compare with IP address and port number from string
    If similar
        Return
Write string at end of mapfile      (reached only if no match in mapfile)
Close mapfile
```

g) Processing a tping request & sending a reply: TPing_reply()

This function is called by NetRecvFunc() when it receives a tping request. It processes the request, and sends back the data required.

```
I32s TPing_reply(ComAddr *cell_addr);
```

The only parameter needed by this function is the address of the creature which sent the request. The return value is an exit status.

The outline of the code for this function follows.

```
Create buffer
Put TPINGREPTAG in buffer
Put local IP address and port number in buffer
Put *cell_addr in buffer
Put SoupSize, NumCells, Speed in buffer
Extract IP address and port number from cell_addr
Fill in destination address for socket with above data
Send buffer contents
```

h) Processing a tping reply: TPing_recv()

This function is called by `NetRecvFunc()` upon receiving a tping reply message. It puts the tping information in the I/O buffer associated with the destination creature.

```
I32s TPing_recv (I8s *buffer);
```

This function receives the data field received on the socket (minus the tag) in its `buffer` parameter. It returns an exit status value.

The outline of the code follows:

```
Extract destination creature from buffer
Put SoupSize, NumCells, Speed in IO buffer for cell
Put source address in IO buffer for cell
```

III.1.3.3-Outline of the network related code

The next few lines will show where exactly the new network code was added in the previous program structure.

The main is apparently unchanged.

```
int main(argc,argv)
int argc;
char *argv[];
{
  GetSoup(argc,argv);
  life();
  WriteSoup(1);
}
```

We saw earlier that `GetSoup()` initialized the soup. The following "code" was added to this function, to create the local communication endpoint and warn the other nodes of the creation of the local Tierra.

```
Call socket system call      (create an UDP socket)
Determine local address
Bind socket to local address
Set socket option to non-blocking I/O

Call Addr_send()
```

The `life()` function gets `NetRecvFunc()` added to its main loop.

```
void life() /* doles out time slices and death */
{
  while(Generations < alive) {
    (*slicer)();
#ifdef NET
    NetRecvFunc();
#endif /* NET */
    ReapCheck();
  }
}
```

The beta version of Tierra had a lot of weaknesses. So a second, “improved”, version was written. It follows the same lines as the previous one, and the functions we presented still exist and (mostly) play the same role.

But this new version, which we will call 1.0, corrected some of the most obvious weaknesses that appeared in the beta version.

III.2-Network Tierra version 1.0

The improvements to the beta version followed three lines. The first one, and least important (from the programmer's point of view, of course) was the modification and extension of the set of message formats available. The second one was the introduction of a new, more convenient method for sending the messages. The third one was the improvement of the management of the list of addresses of remote Tierran nodes.

These three directions for modifications are detailed in the following sections.

III.2.1-New message formats

III.2.1.1-Formal description

The first changes that were introduced dealt with the messages themselves. They followed two lines: modifying existing messages to carry more information, and adding new messages to the preexisting set.

a) Adding to the data brought back by tping

The idea was to add more information to the data sent back in a tping reply. Because we want the creatures to learn how to use this data, and because we do not know what they will consider useful, we can as well fill this with the maximum of information.

The following items were added to the previous ones, which were Speed, SoupSize and NumCells (cf. III.1.2.2):

+ the operating system under which Tierra is running,
+ the total number of instructions executed by Tierra since startup,
+ the total number of instructions executed with Tierra connected to the network,
+ the transit time between local and remote nodes,
+ the local time.

b) Caching tping data

If creatures really begin to use the tping instruction, with a lot of nodes connected, and a lot of creatures going to all the nodes to get their data, the volume of traffic could very quickly grow out of hand. In the worst case, this could bring Tierra down, and in the best case, important messages (such as migrating genomes) could be discarded because of the limitations on buffering size for sockets.

So the second idea was to avoid sending too many tping requests and replies. To do this, we will use a technique called caching, which is commonly used when manipulating frequently accessed data.

Tping data is gathered once, by sending a tping request to the proper site, when the first creature asks for the data for a specific node. The data is then copied somewhere in memory, before being given to the creature which issued the request.

When a second creature asks for the same data, the information will be collected from local memory, instead of sending a message on the network. And so on for the following accesses to this information.

This method drastically cuts down the number of messages being sent over the network. And the access to the tping data is completely transparent to the creatures, which never know whether the data they got came from the network or the cache.

Furthermore, an access to cached memory is much quicker than an exchange of messages over the Internet. This way, creatures willing to go for the net will not be too much penalized by their efforts.

c) Sending tping data out with every message

The problem introduced by caching is to know whether the data in the cache in coherent with the outside world. While being perfectly aware that the data gathered is already out of date when it gets to the local site, it is also clear that we do not want it to be too much out of date.

Keeping the data in the cache relatively up to date would require the local Tierra to send new tping requests from time to time, to every node on the net. This would introduce some unavoidable overhead.

Another method was adopted. To update the caches on the net, each Tierra appends its local tping data to every message it sends. This way the information takes advantage of other messages to travel to its destination, thus avoiding the overhead introduced by sending full messages, processing requests, and sending replies.

d) BYETAG, a new tag for a new message format

After tping requests and replies, we tried to perfect the mechanism used to exchange addresses. A lot of work was (and is still) needed, as can be seen in section III.2.3, but a simple modification was introduced earlier on.

The method we use to teach addresses to other nodes lacked symmetry. We saw that a node, at startup, warns every node on its mapfile that it is awake and ready to receive. But there was no mechanism for a node to warn other nodes that it is leaving the net, or, worse, going down.

This was implemented (at least for cases when Tierra is exiting gracefully: a crash does not leave room for such niceties) by sending a special "good-bye" message to every known node, when going down.

A new type of messages, using the BYETAG tag, was created for this purpose. Its data field simply contains the full address (IP address and port number) of the sender.

Upon receiving such a message, Tierra deletes the corresponding line in its list of node addresses.

e) Summary of the new message formats

EMIGRATETAG	genome (as an array of bytes)	tping data

TPINGREQTAG	source address (creature)	tping data

TPINGREPTAG	source address (node)	destination address (creature)	tping data

SENDADDRTAG	sender address (mapfile format)	tping data

BYETAG	sender address (mapfile format)	tping data

fig 3.4: message formats for Tierra

After implementing the modification discussed in the sections above, we get the set of messages seen in figure 3.4.

III.2.1.2-Underlying structures

To help implement these new message formats more easily, new data structures were added to Tierra.

a) Address structures

We saw that a Tierran node is identified through the association {32-bit IP address; port number}. A structure called `NetAddr` was created to handle such a association:

```
typedef struct {
     I32u node;
     I16u portnb;
} NetAddr;
```

We saw that Tierra also manipulates addresses for creatures. This is done through the ComAddr structure we saw in section III.1.3.1. This structure was modified a little to use the one above.

```
typedef struct {
     I32s ranID;
     I32s cellID;
     I32s soup;
     struct NetAddr address;
} ComAddr;
```

b) Tping data structure

The following structure was created to allow the many different pieces of information appearing in the tping data field of every message to be handled more easily.

```
typedef struct {
  I32s Speed;       /* speed of Tierra */
  I32s SoupSize;    /* size of soup in bytes */
  I32s NumCells;    /* number of cells in soup */
  I8s OS;           /* underlying operating system */
  Event InstExec;  /* total number of instructions executed
*/
  Event InstExecConnect; /* total number of instructions
                            executed while connected */
  I32u TransitTime; /* transit time from source to dest */
  I32u Time;        /* local time at origin */
} TPingData;
```

c) Structure of messages

With the structures defined above, we can define very simply the implementation of the messages formats, as can be seen in figure 3.5.

EMIGRATETAG	char[]	TPingData
	genome	

TPINGREQTAG	ComAddr	TPingData
	source	

TPINGREPTAG	NetAddr	ComAddr	TPingData
	source	destination	

SENDADDRTAG	char[]	NetAddr	TPingData
	name	source	

BYETAG	NetAddr	TPingData
	source	

fig 3.5: Structures and messages

The figure 3.5 also suggests that we could push the idea of using structures a little bit further, and create structures for handling the messages themselves. The ability to send structures over the network would then considerably simplify the exchange of messages.

No one should forget that until now we have been stuck, in order to send anything across the Internet, with the very unwieldy task of translating all data to a stream of bytes, if unlucky, or to and from network standard byte order, if lucky (i.e. if the only types handled were integers).

Unfortunately Berkeley sockets do not provide any mean for sending data in any form but arrays of bytes.

III.2.2-Introducing XDR

The eXternal Data Representation standard was designed to make the exchange of data between heterogeneous machines easier. It defines, as shown in [12], a standard format for data being sent over a network, not only for integers (as the byte ordering functions seen in Annex 2 do), but for most of the data types, simple (integers) or complex (structures, arrays), used by the C language.

This section will briefly explain the use of XDR, and will show how it was used in Tierra.

III.2.2.1-XDR routines

a) Using XDR: the basics

As we saw in the introduction above, XDR propose a formal standard for data being sent over a network. This standard is what can be found in [12].

But XDR is also a library of functions which insure the conversion of data to and from this standard format. This library can be used as it is, or the functions it provides can be used to build other conversion functions for more complex, user-defined data types.

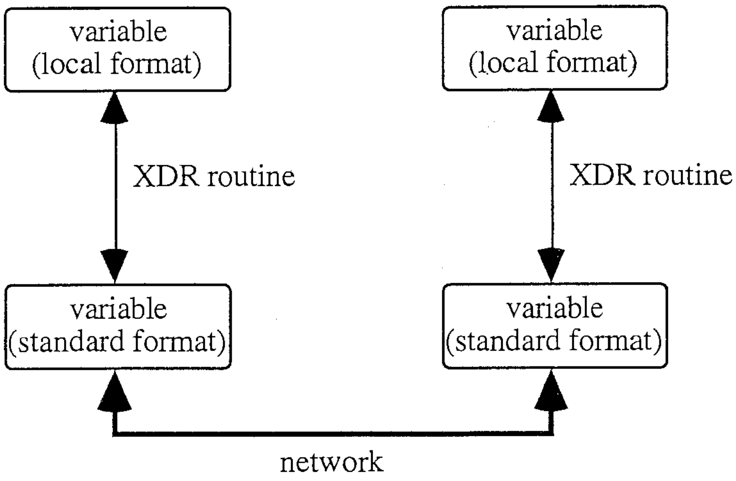

fig.3.6: Encoding and decoding with XDR routines.

When using XDR to send a message, the data composing the message first must be encoded to the standard message format. The different variables making up the message are read, encoded using the XDR routine associated to their type. The result, under the form of a stream of bytes, is stored in a buffer.

It can then be sent across the network, using the standard socket system calls (sento(), send()...).

When using XDR for the reception of a message, the data which was received on the socket is decoded from the standard format. The different conversion routines are called to read byte by byte from the buffer where the incoming message was stored, and write the converted data to local variables of the proper type.

As can be seen in figure 3.6, the encoding and decoding operations are perfectly symmetrical, and are handled by the same routines.

The encoded data is kept in a buffer as a stream of bytes. This data stream is handled through an XDR handle (the definition of the corresponding structure, XDR, can be found in <rpc/xdr.h>). The handle must be created and associated to the buffer before the encoding / decoding operations can begin. The direction of the operation (ENCODE or DECODE) is specified when creating the handle.

Here is a short summary of the way data is sent...

```
Create buffer
Create XDR handle set to ENCODE
While (variable to encode exist)
    Call XDR routine associated to data type for the
        variabe (put encoded data in buffer)
Fill in socket address information (if necessary)
Send buffer contents
```

...and received using XDR.

```
Create buffer
Read from socket to buffer
Create XDR handle set to DECODE
While (unread data in buffer)
    Call XDR routine (put encoded data in local
       variable)
```

It should be noted that XDR does not encode explicitly the type along with the data. The data received in a message encoded with XDR is an unstructured stream of bytes. This means that the user must always know what the data should be decoded into.

It is up to the programmer to keep encoding and decoding operations symmetrical between sender and receiver, and thus guaranty that the data is always coherent.

b) Writing XDR routines

Some XDR routines, which deal with the most used data types (integers, floats, characters), are provided by an XDR library. This library also provides the outline for routines for more complex data types such as arrays or strings. See Annex 2 for more details about these functions.

An XDR routine usually looks like:

```
bool_t xdr_type(XDR *xdrptr, type *param);
```

(the prototype can be more complex for complex types such as arrays...).

XDR routines for user-defined types (structures, unions...) must be written by the user himself. The principle for structures is quite simple. Given the following structure:

```
typedef struct {
  type1 field1;
  type2 field2
  type3 field3
} type4;
```

... the XDR routine associated will be:

```
bool_t xdr_type4(xdrptr,ptr4)
XDR *xdrptr;
type4 *ptr4;
{
  return ((xdr_type1(xdrptr,&ptr4->field1)
           && (xdr_type2(xdrptr,&ptr4->field2)
           && (xdr_type3(xdrptr,&ptr4->field3));
}
```

... where `xdr_type1()`, `xdr_type2()` and `xdr_type3()` are the XDR routines associated respectively with the types `type1`, `type2` and `type3`.

As a consequence, XDR routines for more and more complex data types can be built using the routines previously written for simpler ones, and so on.

For more details on this subject, see the man pages and [13].

III.2.2.2-Automated generation of RPC code: rpcgen

Tierra uses many different structures to build the different formats of messages. To be able to encode and decode messages, we need to write the different XDR routines associated to these routines.

This task is a bit boring, because of the number of structures appearing in messages, and even more so because the formats of the messages should be allowed to evolve along with the needs of the creatures, and the ideas of the users. This means modifying structures and routines every time a small modification is introduced.

To alleviate the tedium of writing XDR routines and modifying them along with structures, the XDR package also provides an automatic generator of XDR code: `rpcgen`. It was used to write the XDR routines in Tierra.

`rpcgen` makes the use of XDR much easier. The programmer only needs to write the outline of the structures in a special file (anything.x), using a syntax roughly similar to the syntax of the C language. `rpcgen` is then run on this file, and creates two files.

The first one, with an extension .h (anything.h), contains the structures previously described, in correct C language, along with the prototypes of the XDR routines. The second file contains the code of the XDR routines, under the name anything_xdr.c.

To modify the structures or the format of a message, the .x file alone needs to be modified. The programmer then simply runs `rpcgen` again on the .x file to update the C structures and the XDR routines.

More details about `rpcgen` can be found in [13].

III.2.2.3-Adding XDR to Tierra

a) XDR streams

This section deals with the XDR stream itself, and the way its creation affects its workings.

The XDR library proposes three routines to create and initialize an XDR stream, i.e. to associate XDR handle and buffer used to store the encoded data: `xdrmem_create()`, `xdrrec_create()` and `xdrstdio_create()`. The last one is used for I/O involving files descriptors, and is of no interest to communications for Tierra. But a choice had to be made between the first two.

It was an easy one to make. `xdrrec_create()` buffers outgoing data until the buffer is full and then calls a routine to send the data, and calls a receiving routine if the buffer gets empty. This behavior is obviously best suited to a connection-orientated protocol.

xdrmem_create(), which stores encoded data in the buffer (gets data to be decoded from the buffer), and requires the application to send the data itself, is more suited to the datagram approach. When sending data, the data segment of the datagram is built in the buffer, and then the send() system call is used. Reciprocally a read() system call puts the data segment of an incoming datagram in the buffer, where it is decoded as needed.

So xdrmem_create() was chosen in our case (as could be deduced from the outline of the way XDR is used that was given above).

xdrmem_create() takes as arguments a pointer to the XDR handle, the address of the buffer, the maximum size for this buffer, and the direction of the XDR operations to be applied to the stream (XDR_ENCODE, XDR_DECODE, XDR_FREE).

Its prototype is:

```
void xdrmem_create(XDR *xdrs, char *addr, u_int size, enum
                   xdr_op op);
```

b) Using rpcgen to generate the necessary XDR routines

rpcgen was used on two files. The first one was a small part of portable.c, where the basic types of integers (8-bit signed, 8-bits unsigned...) are defined independently of the machine. The part chosen was of course the one for Unix based systems. It was used to generate **port_xdr.c**, which contains the code for the following XDR routines, used to translate to and from the user-defined integer types.

```
bool_t xdr_I8s(XDR *xdrs, I8s *objp)
bool_t xdr_I8u(XDR *xdrs, I8u *objp)
bool_t xdr_I16s(XDR *xdrs, I16s *objp)
bool_t xdr_I16u(XDR *xdrs, I16u *objp)
bool_t xdr_I32s(XDR *xdrs, I32s *objp)
bool_t xdr_I32u(XDR *xdrs, I32u *objp)
```

The second file, named **mesg.x**, defines the structures specific to the network communications. It was used to generate the corresponding C structures and XDR code.

The structures defined are the ones we talked about in section III.2.1.2, the basic blocks used to build the different message formats.

But XDR was used one step further, and mesg.x contains the outline of a structure describing any existing Tierran message. It is possible with XDR and rpcgen to create a mix of structure and union where the decision on the type of the members is made by reading a first field (of a fixed type), a tag whose value indicate how the rest of the structure should be interpreted.

This is exactly what we had previously been doing by hand. When called with an XDR stream set to DECODE, the XDR routine associated to this special structure decodes and reads the tag, and, depending on its type, call the adequate XDR routines to decode the rest of the message.

Here is the definition of the message structure which can be found in mesg.x.

```
enum TagsType {
     NULLTAG = 0,   /* introduced for debugging purposes */
     SENDADDRTAG = 1,
     BYETAG = 2,
     EMIGRATETAG = 3,
     TPINGREQTAG = 4,
     TPINGREPTAG = 5
};

union Data switch (TagsType tag) {
    case NULLTAG:
     void;
    case SENDADDRTAG:
        struct AddrBody Addr;
    case BYETAG:
        struct ByeBody Bye;
    case EMIGRATETAG:
        struct EmigrateBody Creature;
    case TPINGREQTAG:
        struct TPingReqBody TPingReq;
    case TPINGREPTAG:
        struct TPingRepBody TPingRep;
    default:
     void;
};

struct Message {
    Data info;
    struct TPingData PingData;
};
```

The XxxBody structures combine the different structures found in the data part of the messages, except for the tping information field (which is part of every message). The details of the structures can be found in figure 3.5 and, for more information, in the file mesg.x (see the code for more details).

c) How to use the code generated by rpcgen

The structures and XDR routines generated by `rpcgen` are stored in **mesg.h** and **mesg_xdr.c**.

We are really only interested in the C structures and the final routine used to encode and decode the `Message` structure.

The C structures need to be understood because they will be used to put messages together before they are encoded, and to store and read the data after it is decoded.

But among the (many) XDR routines generated for these structures, Tierra will only use `xdr_Message()`. This is because this function is called to encode/decode a whole `Message` structure, and calls in turn, by itself, the other routines.

To send or receive a message, the steps to follow are now simply:

```
Declare Message structure
Fill its fields with the data to be send (according
            to the adequate format), tag included
Create buffer
Create associated XDR handle, set to ENCODE
Call xdr_Message() for the structure
Fill in address information for the socket
Send buffer on socket
```

... and:

```
Declare Message structure
Create buffer
Create associated XDR handle, set to DECODE
Read data from socket to buffer
Call xdr_Message() for the structure
Process the data (in the fields of the sructure)...
```

III.2.3-Management of address lists

The management of the address list for remote node is an important point for Tierra, which was not developed as it should have been in the beta version. The improvements added in the version 1.0 were two-fold. First came an improvement of the implementation of the address list itself. But the essential part was the establishment of a global policy for the management of these lists.

III.2.3.1-IPMap array

a) The mapfile and the beta version

The storage of the address list in the beta version was very awkward.

Because the list was stored in a file (the mapfile introduced in section III.1), Tierra had to access the file every time an address was needed, which meant at least every time a message was sent.

The data needed by Tierra could have been stored in a cache after the first access (and stayed there long enough to be useful), thus limiting the access to data on disk. But it was still stored as a sequence of bytes, with a sequential access to the bytes of the file as the only method available for reading the information needed.

To read a single address in a file whose size could grow quite a lot (with the total number of live Tierran nodes on the net as an upward limit), Tierra had to read the file byte by byte, to extract the information line by line, and to translate the string thus obtained into the different integer values wanted (IP address and protocol port number).

This use of the mapfile to store the address list led to a big waste of time. So another method had to be introduced.

b) The IPMap, implementation

The idea is to store the data Tierra wants to get from the mapfile in an array, which would enable fast access at run time, and allow data to be stored in a structured format. This array is called `IPMap`.

The IPMap is a global array of `MapNode` structures, which store the IP address and port number of a Tierran node. Because this IPMap will only be accessed by Tierra, to which names are meaningless (and because a string of no fixed size is difficult to manipulate and eats memory space), the name of the remote node was omitted in IPMap.

However, more information was added to every cell in the array. Some people may have wondered where the TPing data which is now brought to Tierra by each message received is stored. The answer is now clear.

The introduction of the IPMap thus solve the problem of TPing data. This data could not have been stored conveniently in the mapfile, because it is updated frequently. If read-only accesses to a sequential file are cumbersome enough, read-and-write accesses are even worse.

So we get the following for the `MapNode` structure (found in tierra.h).

```
typedef struct {        /* structure for IPMap array */
    NetAddr address;    /* address of remote node */
    TPingData data;     /* Tping data for remote Tierra */
} MapNode;
```

c) Managing the IPMap

A new set of functions had to be introduced to maintain the IPMap. Their prototypes and a short description of their role follows.

```
I8s InIPMap(NetAddr addr);
```

`InIPMap()` is called by the other functions to test whether a given Tierran node (IP address and port number) is in the IPMap. The return value is 1 if the address is already in the IPMap, and 0 if it is not.

```
void AddIPMap(MapNode new);
```

This function adds a new cell, containing the information given in the `new` parameter, to the IPMap array.

It is used at startup to build the IPMap, and during run-time to add previously unknown addresses to the array (for example when receiving a SENDADDR message).

```
I32s RemoveIPMap(NetAddr dead);
```

This function removes the cell in the array with `dead` as an address. It is called when a BYE message is received from the corresponding node.

```
void GetIPMap(FILE *fp);
```

This function is called at startup, to load the data stored in the mapfile into IPMap.

```
I32s SaveIPMap(FILE *old_fp, *new_fp);
```

This function save the content of the IPMap into the mapfile. Because the IPMap must then be totally rewritten, two files have to be opened at that time. The first one (`old_fp`) is the old version, and the second (`new_fp`) receives the updated version.

III.2.3.2-Updating the address list

After studying the mapfile, the IPMap and the functions used to add and remove nodes from them, this section will look at matters from farther away and see how and when these functions are called and the list of addresses updated.

a) Loading the IPMap from the mapfile

The address list is loaded at startup by calling `GetIPMap()`, from the `GetSoup()` function.

For every line of the mapfile, `GetIPMap()` extracts the IP address and port number. Then it fills a MapNode structure with the node address, zeroing the fields for which data is unavailable (meaning the TPingData part). This structure is then used as a parameter for `AddIPMap()`, which adds it to the IPMap.

This procedure is repeated for every node in the mapfile. When all the data is loaded, the mapfile is closed, and Tierra begins using the entries in the array to send messages.

WARNING: It should be noted that while reading the mapfile, the syntax of each line is always assumed to be correct. In theory, the mapfile should respect the syntax rules, because the addresses inside it are written by Tierra itself. If the user was to add an address by himself, and make a mistake when adding it to the file, the GetIPMap() function will systematically abort afterward, causing Tierra to crash...

b) Adding an address

When Tierra receives an address, it checks whether it was previously known or not. If it is a new one (i.e. if the node is not in the IPMap), it has to be added to the array. This is done in two ways, depending on the way the information was received.

The most obvious way is for the address to arrive in a SENDADDR message. The IP address and port number are extracted, and AddIPMap() is called for this address. The new address is now in the IPMap.

This is enough for Tierra. Unfortunately, IP addresses do not mean much to human users. This is the reason why the actual name of the remote site was added to the message. But the IPMap, as we saw previously, does not deal with names... One solution would have been to simply not bother with them, and contact the Domain Name Server to get the information whenever the user asked for it, or when saving the IPMap to file (querying the name server can take some time).

Another solution was chosen. When a new address arrives with the associated name, the mapfile is opened and the whole data, address and name, is written at its end.

Sometimes a message other than SENDADDR can arrive from a previously unknown site. In this case there is no name associated with the address. The AddIPMap() function is called with the appropriate parameter, and the new node is added to the IPMap.

So the data is treated differently depending on the kind of message carrying the new address. This means that the mapfile is only partially up to date at any time. The following sections will show how it can grow more and more incoherent as Tierra keeps running, and will study the consequences.

c) Removing an address

When Tierra receives a BYE message signaling that a node is going down, it erases the corresponding cell in the IPMap array by calling `RemoveIPMap()`. No more messages will be sent to this node.

Because the node is erased from the IPMap and not from the mapfile, the two address lists become more and more different. The fact that new sites whose names are known are added to the mapfile makes the situation worse.

We suppose that the file is coherent when Tierra first downloads its contents. As new addresses keep arriving, the IPMap is updated, but the mapfile registers only part of these updates. If a node present in the mapfile goes down gracefully, and broadcasts a BYE message, it will be erased from the IPMap. But not from the mapfile. Suppose now that the node goes back on line. It then broadcasts a SENDADDR. When this is received, the address is tested against the IPMap. Because it won't be found there, it will be added to the IPMap... and to the mapfile. Because the previous reference was not erased, this node will be referenced twice in the mapfile. Which will be incoherent.

The first conclusion is that the user should never rely on the mapfile at run time.

d) Updating the mapfile

The previous lines showed that there is a big need to update the mapfile. This is done by Tierra every time the soup is saved to disk, inside the function `WriteSoup()`. This calls the function SaveIPMap().

SaveIPMap() gets two open file descriptors, one to the mapfile, the other to a temporary file (called "ipmap.tmp", so don't call any file by this name in your Tierra directory!). The function then reads the mapfile. At every line, it extracts the address, and checks it against the IPMap. If it is absent, it does nothing. If it is in the IPMap, it writes the line, name included, to the temporary file. And it puts a tick somewhere to remember this node is in the new mapfile. In case the reference is in the IPMap, but was already ticked, it does nothing.

After going through the mapfile, the function goes to the IPMap, and look for any unticked cell. If it finds one, it writes the address in the temporary file, with "name unknown" in the name field.

Then WriteSoup() close the files, delete mapfile, and rename the temporary file to the name previously given to the mapfile. Tierra now has a coherent, up to date mapfile, that the user will be able to consult with confidence in the reliability of its contents.

Things look to be perfect. Except for a small detail: what about the times when Tierra goes down in an uncontrolled fashion? The current philosophy, valid for the soup, is that this it is not worth fretting about. The results of the last run are lost, but the old soup is still secure on disk. However, the mapfile is modified by Tierra, and can be not only outdated (which would not be too bad, same as for the soup), but corrupted enough to be incoherent. When it is loaded again at the next start, the IPMap itself will be incoherent (double references to a node...), and this time even a controlled stop of Tierra will not solve the problem.

There is currently no solution to this problem. To protect the data in the mapfile (and avoid the need for the user to fiddle with the file by himself), a backup copy should be made before every run of Tierra...

The previous sections dealt with the nitty-gritty details of address transmission. But nothing was said about a global scheme to insure that the data in all the different mapfiles on the network is coherent.

This is to say that version 1.0 works, but leaves a lot of problems untreated, the addressing scheme being only one of them. The next section exposes a few of these problems, and proposes a few directions where to look for solutions.

III.3-Toward version 2.0

This section will discuss a few of the problems that were not solved in version 1.0, or that appeared when the new version was implemented. Some of the problems that appeared are rather simple to solve.

The first one would be the mess in the variables used by Tierra. The previous sections might have been misleading, but the programmation of the features detailed there was much messier than shown, especially with three people working on the many files. I am pretty sure a lot of the variables that where created to handle the beta version are now obsolete, or redundant, or even incoherent with later ones.

Other small problems and possible improvements are detailed in Annex 3.

However, some of the problems are definitively not simple. This section deals with three of them. In every case the problem is explained, and whenever possible some kind of solution is suggested.

However, I don't guaranty that the solutions given will be the best, or will even work...

III.3.1-A coherent scheme for address transmission

III.3.1.1-Problem

The first problem deals with addresses and more precisely the global scheme for address transmission and update. In section III.2 we studied extensively the IPMap, the mapfile, and the details of address transmission.

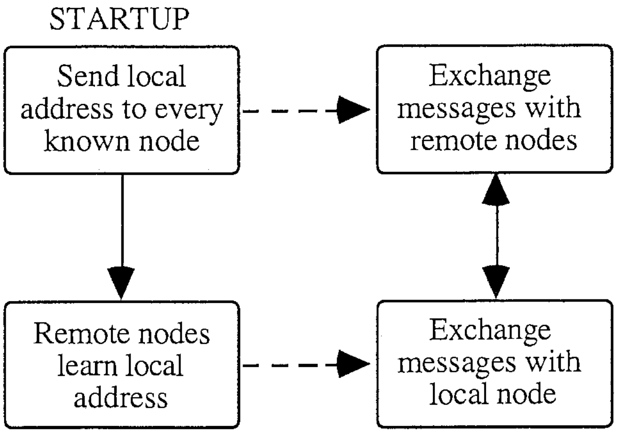
STARTUP
Send local address to every known node
Exchange messages with remote nodes
Remote nodes learn local address
Exchange messages with local node

fig. 3.Z: Address and message exchanges

In the present version of Tierra, the address exchange scheme is the one shown in figure 3.Z. Part of the problem is obvious: there is no way Tierra can exchange any message with other nodes if there is nothing in the local mapfile.

Not only this, but Tierra has no place to which it can send its own address at startup. As a consequence, none of the remote nodes will be able to reach it, or even to know it exist. Tierra will never receive any message, and thus will never learn new addresses to fill its (empty) IPMap.

The problem is even more complicated. Let's suppose that the local Tierra knows the address of a few nodes (the user stepped in and wrote those in the mapfile, deus ex machina).

Now a new node appears on the net. It also knows the address of a few nodes, some of which are in the list of the local node. In this case, it will be able to send its address and to exchange messages with the nodes it knows. But the local node is not on this node's address list. The local node never received the first message broadcasting the new address, and because it never learned this address, it will never send messages to this node. Which in turn will never learn the local address, etc...

We get the same problem as with no addresses in the mapfile, one step added.

The conclusion is that we created a complicated system to allow the nodes to exchange messages, but that, because of the lack of coordination between the nodes, this wonderful system will never be used. Even worse, if the user does not put a few addresses in the mapfile before running Tierra, no messages at all will be exchanged. To summarize the situation, we have a wonderful network version of Tierra... which will never be able to use a network!

III.3.1.2-Solution

The coordination between nodes presents a crucial problem for address exchanges, but can be solved rather easily. The features to be added to Tierra to get a workable network of Tierran nodes (out of our workable network Tierra) goes along two lines.

a) A central "address server"

First, we need every node to begin running with a non-empty mapfile. This will take care of the first part of the problem.

No modification needs to be made on the code at this point. Tierra should simply be distributed with a mapfile containing the address of one node. However, we need to be certain that this node will be up and running, and thus able to receive messages, when the local Tierra broadcasts its address.

So we need at least one machine on the Internet where we can be sure the network version of Tierra is running and able to receive messages at any time. Two machines (or more) would be better, to allow for a backup in case one machine is down, or even isolated from the network by a problem on the local network (disconnection) or on the Internet (loop in the routing tables, overloaded links...).

b) Address requests

Now that we have a starting point on the net (namely this central name server), the local Tierra should be allowed to learn new addresses. If a new mechanism is not introduced, Tierra will be stuck with the first (or first few) machine(s), and furthermore these machines would quickly overload as all the messages would be directed to them.

A few modifications should be made to Tierra itself, to allow a node not only to broadcast its own address, but also to send one address already in its mapfile to a remote node.

A new message format is needed, for a "request for addresses". Let's call it ADDRREQ. When a node receives such a message (add the appropriate functions to process this new message format), it sends back a SENDADDR message for every node in its IPMap. The destination node can then add all these addresses to its own IPMap.

c) General scheme

We now have a mechanism to learn addresses. We have a central server giving a starting point on the network.

The new message should be the first one sent to the central machine, to learn other addresses. (Upon receiving a message, a node always tries to add the sender to its IPMap, so we can skip the first SENDADDR message when contacting the address server at startup).

After this first step, there are many different ways for the machine to try and increase the amount of data in its IPMap. Contacting known nodes to get the content of their IPMap is a good idea. But this could very easily lead Tierra to spend 99% of its time playing with new addresses, sending ADDRREQ for every new address, receiving an increasing amount of messages, with a diminishing amount of useful information.

The other extreme would be to contact the address server, download its IPMap, and stop here with ADDRREQ. But what about the first node to contact the address server? It will get no new address, and will have to rely on the other nodes, the ones that will contact the server later (and thus get its address), to send a message. Because in this case no further explicit address exchange is done by any node after contacting the server, the first node can only pray that the creatures on the other ones will be very interested in network communication...

So the general scheme for Tierra should be somewhere between these two extremes, perhaps using timers on the IPMap to limit the number of ADDRREQ messages without restricting it to one (the one to the address server).

After a few runs, Tierra should have stored enough new addresses in its mapfile not to need the server so much, even if the data is in part outdated when Tierra begins a new run. So perhaps the central server could be bypassed in this case to avoid overloading it.

The number of solutions is unlimited.

III.3.1.3-Discussion

Some people could object that setting up a central machine introduces a hierarchy between machines where none should logically exist. This is not really true.

First, this "central" Tierra is only central for addresses. There is no hierarchy introduced for the creatures, and the messages they exchange.

Furthermore, there is no other way to teach a node its first address (the practical reasons are always the best). Because there is no logical difference between having a central server on a machine somewhere on the net and having an all-knowing user handing out the addresses.

This (a central server for a set of information) is more or less the method used by the Internet to handle machine names. See [10] and the chapter about the DNS for a discussion of this point.

III.3.2-Polling, an inefficient method?

III.3.2.1-Problem

As soon as version 1.0 was implemented and tested, it appeared that it was running much more slowly than the current non-network version. So the network functionalities are slowing Tierra down quite a lot. Precise measurements of speed should be done, disabling the `nice()` function in the network version, or adding it to the non-network one, to be fair and have both processes running at the same priority. But this was not really unexpected.

The network communications in Tierra introduce a lot of overhead, especially so because we used polling to get the messages. This simply means that Tierra checks periodically the socket for incoming data. As can be seen when enabling error messages on the `recvfrom()` system call, most of the time this is done for nothing. Polling is a very expensive method for data communication.

However, polling had to be introduced (see III.1) because we cannot allow Tierra to block when reading data on the socket.

Furthermore, because there is a limit on the amount of data a socket can buffer, and because some messages (namely emigrating genomes) could get quite big, the length of time between two calls to `recvfrom()` should not be too long. Or two big messages coming one shortly after the other could cause data loss.

So decreasing the frequency of the checking would not be a good solution.

And the problem is even more important that the slowdown observed was a minimum: at this time, Tierra does not have a creature which is be able to use network communications. So this is a raw overhead, for administrative tasks only.

III.3.2.2-Solutions

The solutions to this problem are difficult to find. Polling was convenient for a lot of reasons, and a replacement would have to provide at least as much reliability.

An idea would be to replace polling by asynchronous I/O communications. The Unix kernel sends Tierra a specific signal (SIGIO, for incoming I/O) anytime data is available on a descriptor belonging to Tierra.

Instead of ignoring the signal (which is its default behavior), Tierra would catch it, meaning it would call a function specified at startup (namely `NetRecvFunc()`) to do whatever is appropriate with the incoming data.

This presents a few drawbacks. First, signals are expensive to catch, because they involve a lot of interaction between process and kernel. So we would perhaps not get much better performances than with polling. This should be tested...

Furthermore, Tierra actually uses more than one socket. The different observational tools that allows the user to watch Tierra running use sockets for communications. These sockets are distinct from the one used by Tierra for network communications (especially as they are TCP sockets), but the kernel delivers the same signal whether it is for one socket or the others. So all the sockets would have to be checked for incoming data. Other problems are introduced by signal blocking: should SIGIO be blocked when handling one occurrence? If it is the case, another occurrence could be ignored when treating the first one... And Tierra currently blocks signals for some operations.

Sockets offer special mechanisms for such problems, but they imply that Tierra blocks on socket descriptors when no data is available.

Further thinking on this problem should begin with [11], sections 6.13 and 8 (for socket specific problems) and 2.4 (for more general information on signals).

III.3.2.3-Discussion

The discussion will be simple: is the network version really that much slower? Is this slower pace not in part due to the high rate of forced expulsion for genomes? Or to the load added by processing tping data every time a message arrives?

Is polling, which is known to waste CPU time, really responsible for all the delays? Or could some optimization be done on the rest of the code? Because changing the principle of communication will obviously not get rid of all the delays: everything comes at a cost.

So very careful testing of both the current network version against the non-network one, and polling versus signal-driven communications should be done, to pinpoint the problem and possible solutions.

III.3.3-Genomes, toward more reliability

III.3.3.1-Problem

Tierra uses UDP, an unreliable, best effort packet-delivery protocol, to send its messages over the network. UDP gives very good results on a reliable underlying network such as an Ethernet, but these results can get less than satisfying when using an unreliable support such as the Internet (which provides a service at most as reliable as the least reliable network on the way from sender to destination machine). This basically means that many messages sent by Tierra could get lost, without the nodes being aware of the situation.

When making a choice for UDP at the very beginning of the implementation of the network version (cf. III.1.1), it was considered that this would not really be a problem. Creatures in the soup are working in an unreliable world already (a world where the CPU can hand out wrong results and where the memory gets periodically corrupted at random). Unreliability (such as mutation) is a necessary feature for the evolution of the creatures. So a little more unreliability does not really harm them.

The only drawback is that the user cannot control the rate of errors on the physical network the way he controls the mutation rate for the creatures. However, network unreliability provides a virtual network with a feel of greater reality.

But creatures are not the only entities using UDP for communication. Tierra itself has to send the necessary administrative messages (address diffusion...). And some of these messages can be critical (the first message asking for the contents of the IPMap on the node server, for example). Furthermore, messages carrying genomes are considered important enough that we do not want the network to loose (too many of) them.

So Tierra basically has a need for more reliability than UDP can provide, more reliability than expected at first. The results are good enough on a LAN (Local Area Network), but the performances out in the real world (meaning for example between two machines on two different continents) would probably be catastrophic.

III.3.3.2-Solutions

The first and most obvious one would be to go from UDP to TCP, which provides a totally reliable service. But the objections to TCP are still the same as in section III.1. It would slow down Tierra far too much, a concurrent server would need the creation of child processes, an iterative one would spend too much time on communication overhead compared to the size of the messages sent by Tierra...

So we are stuck with UDP (meaning we made the right choice the first time).

The problem is now to add reliability to UDP. This is done by adding (by hand) some of the features of TCP to UDP. These new features should be timing and re-transmission for outgoing messages, and the use of acknowledgements for incoming ones.

To avoid adding too much overhead to Tierra, these mechanisms should only be used when really needed. Namely:

+ For EMIGRATE messages, acknowledgement, timing, and re-transmission. As said before, we want do everything we can to make sure these messages arrive safely to destination.

+ For TPINGREQ messages, timing and re-transmission. Or, using the same method as the actual Unix "ping" command, send a fixed number of requests at a time, and use the answer(s) as an acknowledgement, or the lack of answer as proof the remote node is unreachable (updates TPingData in IPMap with "infinite" transit time and invalidates other fields?). If many answers arrive in a fixed amount of time, the values brought back should be averaged. If one wants to be really serious about the unreachable node part, perhaps there is a way to use the ICMP messages raised in case of network failure along the way...

+ For TPINGREP messages, nothing, the other side takes care of the problem.

+ For BYE messages, nothing, they are not essential. The next TPINGREQ will mark the node as unreachable, as would be the case if Tierra had not gone down gracefully.

+ For ADDRSEND messages... probably nothing.

+ For ADDRREQ messages (should the format ever be implemented), probably the same mechanism as TPINGREQ messages, using ADDRSEND as an equivalent to TPINGREP.

Because Tierra cannot wait for an acknowledgement or an answer to its messages, a mechanism must be provided to match replies and acknowledgements to messages sent in the past.

The first step would be to add a message ID number to a request. (Request messages are EMIGRATE, TPINGREQ and ADDRREQ, from what was said above). The associated answer (or acknowledgement) would carry back the same ID number, allowing the local node to match it to the request.

In order to insure the matching of request and reply, Tierra must keep a queue in memory with, for every outgoing request, a structure containing the message ID number and type, and a TTL (Time To Live). Plus the data field of the message, if re-transmission is a possibility (e.g. for EMIGRATE messages).

If a satisfying reply arrives for a message recorded in the structure, the reply is processed and the message erased from the queue.

The TTL is used as a timer on the message and is periodically decremented. If it reaches zero before a reply arrives, the adequate policy is applied (re-transmit message, or consider the remote node as unreachable, or do nothing...), and the entry is removed from the queue (or, in case of re-transmission, the TTL reset to its maximum value).

Any reply message arriving on the node with an ID number that does not match any of the entries in the queue would be discarded.

However, for messages such as TPINGREQ or ADDRREQ for which different answers are possible, the best policy would be to keep treating answers as long as they keep arriving and the TTL is greater than zero.

For more details about the implementation of reliability inside TCP and the outline of a mechanism of acknowledgement and re-transmission, see [10] and the chapter about TCP.

IV-Conclusion

Tierra can now run on a network, in a limited way.

However, there is a lot of work left. Chapter I exposed the goals of the project currently under work, and these have only partially been reached.

Most of the work that was done during the time I spent in ATR dealt with the interface between Tierra-the-application-program and the outside world. This was the easiest part, and the one on which further work should be built (after the questions raised is section III.3 have been taken care of).

But it is far from being enough to get a viable and hopefully dynamic ecosystem running in cyberspace.

The virtual network simulated by Tierra is very primitive. It does not allow a lot of freedom to the creatures, which have very few ways to use it, or even really realize they are in a environment different from the non-network one. For now they only have one new instruction to play with (the "tping" one).

As the Tierra group had a few discussions on this subject, the reader will not be spared a few of the ideas on what could (should?) be done.

The possible needs of the creatures should be carefully assessed, and the adequate instructions added to the new "networked" instruction set. Part of the work will be to decide what should be given to the creatures to do, and what is better left to Tierra.

For example Tierra now uses the tping instruction through a new specialized register in the cell's CPU, which point to the current IP address in the IPMap. But as the IPMap is updated, the address pointed to by this register might change without the creature being aware of it. But should creatures really play around with IP addresses?

One type of instruction that could be useful would be one allowing creatures to communicate. Some of the work now being put in the I/O system between creatures could be transposed to communication between creatures on different nodes. This new mechanism could be kept distinct from the local I/O, or made transparent to the creatures, depending on the choice of the programmer.

Another problem will be to encourage the creatures to actually make the effort to go on the net. Because communications over the Internet will be much slower than Tierra itself, creatures will have to be able to wait for incoming replies to a request. Perhaps a new flag could be added to the CPU structure, signaling the creature in blocked on reading and is waiting for I/O. Creatures will have to chose nodes on networks that are close to them (in access time) for communication to avoid blocking on I/O until they the time they die...

To summarize the previous lines, this report and the associated code only provides a sketchy basis for the real thing.

To all the people that will be working on the next part, good luck, and have fun! I certainly enjoyed myself doing the first part. Well, most of the time anyway.

And there is still the network ancestor left to write...

ANNEXES

Annex 1: Client-server models using sockets

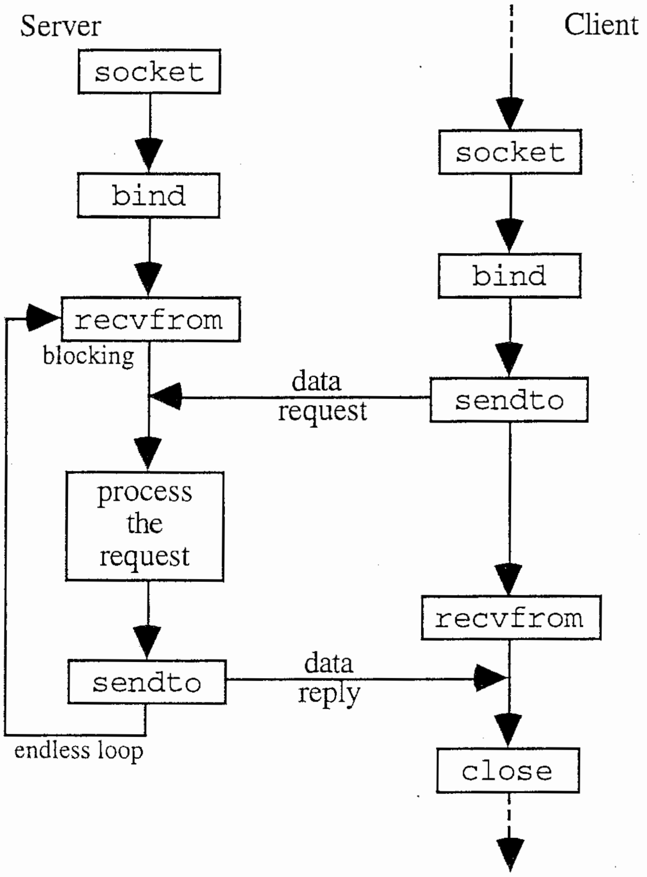

fig A1.1: iterative server and datagrams (UDP)

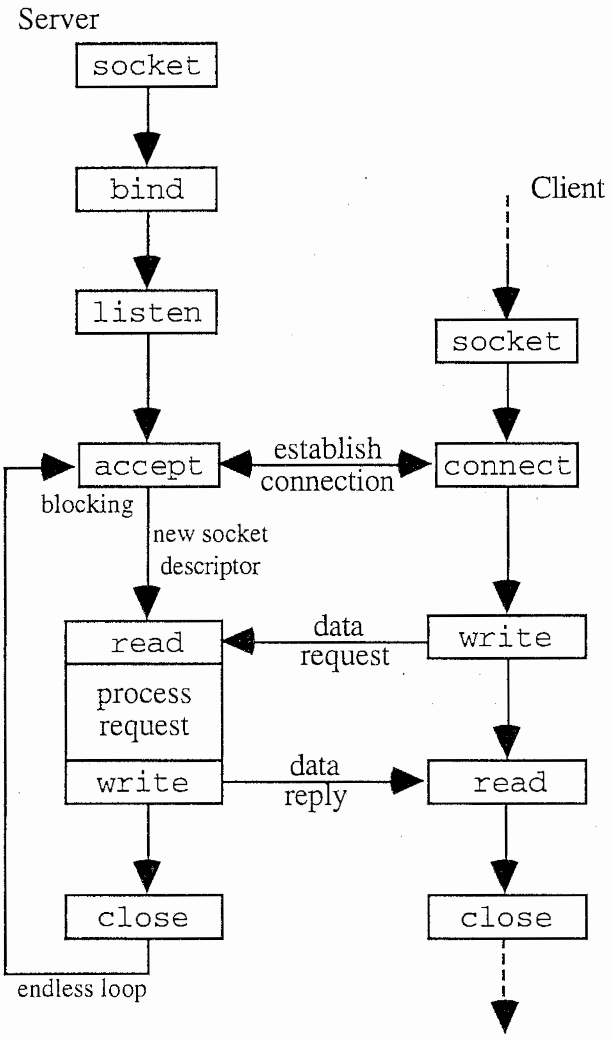

fig A1.2: iterative server and streams (UDP)

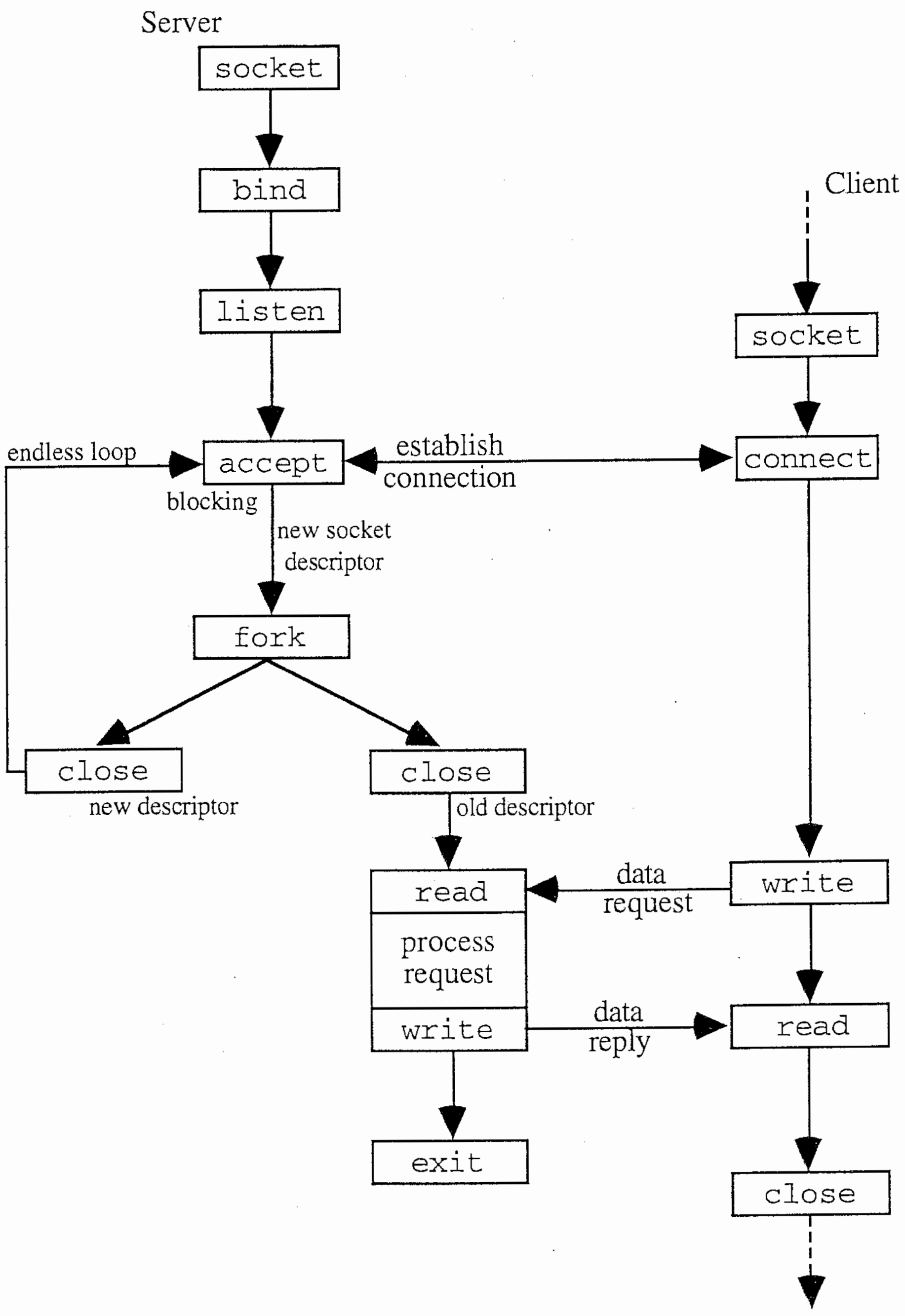

fig A1.3: concurrent server and streams (TCP)

Annex 2 : Socket system calls

This annex only deals with Internet specific information. For other protocols (XNS and Unix domain protocols), see [11].

The types used in the code are:

char	=	8-bit signed integer,
u_char	=	8-bit unsigned integer,
short	=	16-bit signed integer,
u_short	=	16-bit unsigned integer,
int	=	24-bit signed integer,
u_int	=	24-bit unsigned integer,
long	=	32-bit signed integer,
u_long	=	32-bit unsigned integer,

and are defined in <sys/types.h>.

A2.1-Socket address structures

This structure is defined in <sys/socket.h>.

```
struct sockaddr { /* generic structure for a socket address
*/
  u_short sa_family;  /* address family: AF_XXX value */
  char sa_data[14];   /* up to 14 bytes of protocol specific
                         address */
};
```

These structures are defined in <netinet/in.h>.

```
struct in_addr { /* internet address */
  u_long s_addr; /* 32-bit netid/hostid, net byte order */
};

struct sockaddr_in { /* sockaddr structure for Internet */
  short sin_family;  /* AF_INET */
  u_short sin_port;  /* 16-bit port nb, net byte order */
  struct in_addr sin_addr; /* internet address */
  char sin_zero[8];  /* unused */
};
```

A2.2-Elementary socket system calls

The code for the following system calls is located in <sys/socket.h>.

A2.2.1-Creating a socket

a) Socket system call

The socket system call specifies the type of communication protocol desired, and returns a small integer socket descriptor. It creates the socket.

```
int socket(int family, int type, int protocol);
```

where family = AF_INET, type = {SOCK_STREAM; SOCK_DGRAM; SOCK_RAW}; protocol is usually set to 0, because the combinaison of protocol family and protocol type is usually enough to define the protocol. However, the valid combinaisons with protocol ≠ 0 are the following:

{SOCK_DGRAM; IPPROTO_UDP} Ł UDP
{SOCK_STREAM; IPPROTO_TCP} Ł TCP
{SOCK_RAW; IPPROTO_ICMP} Ł ICMP
{SOCK_RAW; IPPROTO_RAW} Ł raw

The values of the IPPRTO_XXX are in <netinet/in.h>.
It should be noted that to use sockets with a SOCK_RAW type, root privileges are requested.

b) Bind system call

The bind system call assigns a name (or more precisely a protocol port number) to an unnamed socket.

```
int bind(int sockfd, struct sockaddr *my_addr,int addrlen);
```

where my_addr is a pointer to a protocol specific address structure (this means that a cast is necessary to make it into a pointer to the generic address structure); and addrlen the length of this structure.

There are three uses to this system call. A server for a well-known service calls bind to register its well-known address with the system. A connection-oriented client can register a specific address for itself (this is not requested, see the connect system call for more details). A connectionless client uses it to insure that it is affected an unique address.

A2.2.2-Readying a socket for communication

a) Connect system call

The connect system call connects a socket descriptor, meaning that it establishes a connection with a server.

```
int connect(int sockfd, struct sockaddr *servaddr,
            int addrlen);
```

The parameters are similar to the ones for bind (and one should not forget to use a cast for servaddr).

For a connection orientated protocol (i.e. TCP), this call results in the actual establishment of a connection, and only returns when it is established.

A client does not have to bind a local address before calling connect: the establishment of the connection causes both remote and local addresses to be assigned.

For a connectionless client, calling connect insures that only messages coming from or going to the specified address will go through the socket. It also allows notification of an invalid address. Last, but not least, it allows the programmer not to bother again with the destination address when using the socket for read and writes.

b) Listen system call (stream only))

The listen system call is used by a connection-orientated server to indicate its willingness to receive connections.

```
int listen(int sockfd, int backlog);
```

backlog specifies how many connection request can be queued by the system while it waits for the server to execute an accept system call (the maximum is 5, and it is also the usual value). This value is most important for an iterative server.

c) Accept system call (stream only)

The accept system call has the server waiting for a connection from a client process.

```
int accept(int sockfd, struct sockaddr *peer, int addrlen);
```

The peer and addrlen arguments are used to return the address of the client. addrlen is a value-result argument, initially set to the size of *peer.
The integer returned is either an error indication or a new socket descriptor on which the incoming data will be put.

A2.2.3-Sending and receiving data

a) Send, sendto system call

These two system calls are similar to the standard write on a file descriptor, except that a socket requires additional arguments.

```
int send(int sockfd, char *buff, int nbytes, int flags);
int sendto(int sockfd, char *buff, int nbytes, int flags,
           struct sockaddr *to, int addrlen);
```

buff is a buffer containing the data to be sent, nbytes is the number of bytes to be sent.
The flag argument is either 0 or formed by or'ing MSG_OOB (out-of-band data), MSG_DONTROUTE (bypass routing).

For sendto, to specifies the address where to send the data (should use a cast), and addrlen is the length of this protocol specific address.

Both returns the number of bytes actually sent (on success).

a) Recv, recvfrom system call

Again, these two system calls are similar to the standard read on a file descriptor.

```
int recv(int sockfd, char *buff, int nbytes, int flags);
int recvfrom(int sockfd, char *buff, int nbytes, int flags,
           struct sockaddr *from, int *addrlen);
```

The arguments are the same as the one for send and sendto.
buff will obviously receive the data contained in the message. For flag, MSG_PEEK means "peek at incoming message" (i.e. do not discard message after returning from system call).

For recvfrom, the from argument returns the (protocol specific) source address of the message received.

A2.2.4-Closing a socket

a) Close system call

The normal Unix close system call is also used to close a socket.

```
int close(sockfd);
```

It should be noted that if the protocol used is reliable, the kernel will have to keep trying to send buffered data after the close returns.

b) Other cases

If close is not called, the socket descriptor will automatically be closed by the kernel when its owner process exits (just as any file descriptor).

A2.3-Useful routines

A2.3.1-Byte ordering routines

Because different machines encode integers using different formats, a standard format was introduced for network communications. The following routines handle this potential byte order difference between source and destination by converting integers to and from this standard format.

```
#include <sys/types.h>
#include <netinet/in.h>

u_long htonl(u_long hostlong);      /* host to net, long */
u_short htons(u_short hostshort);   /* host to net, short */
u_long ntohl(u_long netlong);       /* net to host, long */
u_short ntohs(u_short netshort);    /* host to net, short */
```

A2.3.2-Address conversion routines

The following routines convert IP addresses between 32-bit integer format and dotted decimal format (stored as a string).

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(char *ptr);
char *inet_ntoa(struct in_addr inaddr);
```

A1.3.3-Byte operations

To manipulate user-defined byte strings (i.e. not ended by NULL), such as the ones in the address structures:

```
int bcopy(char *src, char *dest, int nbytes) /*copy nbytes
                                          from src to dest */
int bzero(char *dest, int nbytes) /* writes nbytes NULL to
                                     dest */
int bcomp(char *ptr1, char *ptr2, int nbytes) /* compare two
                         strings, return 0 if identical */
```

A2.4-Reserved port numbers

There are two ways for a process to bind an Internet port number to a socket. The first one is to ask for a specific port number, which can be granted (if it is available) or not by the system. The second one is to ask the kernel for any unused port number, which will be chosen in a specific range.

Unix also supports the concept of reserved ports, which are the well-known port numbers (for standard applications such as ftp, echo...) and a few others reserved for the use of the super-user.

So we get the following port numbers and their possible use:

	Port range
reserved for standard applications	1-255
reserved by 4.3 BSD Unix	256-511
to be used with rresvport function (root)	512-1023
automatically assigned (user clients)	1024-5000
others (user servers)	5000-upward

Annex 3: Toward version 1.1

This annex gives a list of a few of the "simple" things that should be introduced or modified to improve the present version of Tierra. They are considered "simple" because the modifications implied are pretty straightforward, and less open to controversy than the bigger modifications suggested in section III.3.

Some were already mentioned (in the chapter III of the main text), others are new.

A3.1-Local protocol port number

The relevant code was not detailed, but at startup, when the local Tierra creates its socket, it gives a protocol port number to be associated with the socket. This port number is then used to identify Tierra in all subsequent communications. It is the number that is part of the mapfile entry for the local node.

In version 1.0, Tierra always gives 8001 as a port number argument to the `bind` system call. This is in the non-reserved range traditionally used for user defined servers. Everything seems all right.

However, if 8001 is already affected to another application (for example another Tierra running on the same machine), the `bind` system call returns with a non-null status, and Tierra exits. As long as the other application runs with its socket open with 8001 as a port number, Tierra will not be able to even go through startup. This can be a big problem, especially on multi-user machines.

The solution to this problem is pretty simple.

The bind should be included in a loop. As before, Tierra first try to bind to 8001 (or 5001, the first non-reserved value for user defined servers). As long as bind returns with a non-null status, Tierra increments the port number value with 1, and calls bind with this new value. Because the number of processes running on a machine is limited, and because the upper limit on the port number (16-bit unsigned integer) is in theory quite high, Tierra should be able to find a valid port number before reaching this limit...

A3.2-Mapfile

We saw in section III.2.3 that in case Tierra crashes (or exits using `FEError()`) during a run, the data in the mapfile can be corrupted. The solutions to this problem is two-fold, and was also outlined in the main text.

First, a backup of the mapfile should be made at startup, before actually reading the file to create the IPMap array. This backup should have an unique "well-known" name, because any temporary name would be lost in case of a crash. This way the previous state of the mapfile is preserved.

If Tierra goes down gracefully, the function treating its exiting should remove the backup from file.

This way, every time Tierra begins a run, it first checks for a backup mapfile. If one exists, Tierra delete the mapfile, and copy the backup to the name of the mapfile. It can then create the IPMap from the old mapfile data. If no backup mapfile can be found at startup, the mapfile is copied to the backup file, and the IPMap is created from the mapfile.

A3.3-IPMap and frontend

We saw in section III.2.3 that we wanted to avoid having the user rummaging in the mapfile for information about the possible destination addresses, especially so at runtime (if Tierra wants to access the file while the user is reading it, even in read only...).

Furthermore, there is no way for the user to really know what nodes Tierra is really talking to: the mapfile is accessible but never really up-to-date.

To solve both problems in one stroke, it should be a good idea to add the content of the IPMap to the data available though the frontend. Tierra currently displays the number of nodes in the IPMap. Along with the genome histograms, the users could have a menu option displaying the nodes in the IPMap, perhaps as addresses only, at first. Of course, there would be no names.

As a second option, Tierra could display the TPing data or part of it (Soup, Speed and NumCells), which would give a good idea of the state of the different nodes on the net.

Bibliography

[1] "An Overview of Evolutionary Computation", W. M. Spears, K. A. De Jong, T. Bäck, D. B. Fogel, H. de Garis, 1993.

[2] "Artificial Life: The Coming Evolution", J. D. Farmer, A. d'A. Belin, in "Artificial Life II", Santa Fe Institute Studies in the Sciences of Complexity Proc. vol. X, Addison-Wesley, 1991.

[3] "Artificial Life", C. G. Langton ed., Santa Fe Institute Studies in the Sciences of Complexity Proc. vol. IV, Addison-Wesley, 1989.

[4] "Artificial Intelligence Through Simulated Evolution", L. J. Fogel, A. J. Owens, M. J. Walsh, Wiley Publishing, 1966.

[5] "Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution", I. Rechenberg, Frommann-Holzboog, 1973.

[6] "Adaptation in Natural and Artificial Systems", J. H. Holland, The University of Michigan Press, 1975.

[7] "How I created life in a virtual universe", T. S. Ray, 1992.

[8] "Evolution, Ecology and Optimization of Digital Organisms", T. S. Ray, 1991.

[9] "A proposal to create a network-wide biodiversity reserve for digital organisms", T. S. Ray, 1994.

[10] "Internetworking with TCP/IP vol.I", Douglas E. Comer, Prentice Hall.

[11] "Unix Network Programming", W. Richard Stevens, Prentice Hall.

[12] "RFC 1014", Sun Microsystems Inc., Network Information Center.

[13] "SunOS Network Programming Guide", Sun Microsystems Inc.

[14] "Internet", Bruce Sterling, The Magazine of Fantasy and Science Fiction, February 1993.

<u>Note</u>:

All papers by Tom Ray available by anonymous ftp from tierra.slhs.udel.edu in directory /tierra/doc/.

All RFCs available by anonymous ftp. See Annex 3 in [10] for a list of the sites where they can be found.