Master Thesis
University of Neuchâtel

P-ICE
"Peer to Peer" Inexpensive Computing Environment

Viceić Predrag, viceic@net2000.ch
Thesis director: Dr. Sviatoslav Voloshynovskiy,
CUI, University of Geneva

30th July 2002

"From the moment when the machine first made its appearance it was clear to all thinking people that the need for human drudgery, and therefore to a great extent for human inequality, had disappeared. If the machine were used deliberately for that end, hunger, overwork, dirt, illiteracy, and disease could be eliminated within a few generations."

George Orwell in "1984"

To Mila and Anne.

As this master thesis is a result of a collective effort, I would like to thank:

Dr. Alexander Herrigel, for his extensive knowledge in message passing/queuing mechanisms, the fruitful creative environment, computational base and problem formulation.
"Are you sure that the complexity of installation process is a good argument against CORBA?"

Michel Paschalides, for his support in obtaining necessary hardware,
*"Do you *really* need all those Peers?!"*

Olivier Evalet, for the adaptation of the watermarking algorithm for JNI interfacing,
"Honestly, I think I have done enough work on this ~!!!@ DLL!"*

Quentin Brossard, for the administration and configuration of Peer machines,
*"Until I Am the network administrator you *will* put a password on your Admin account!"*

Marc Kalberer, for the coffee break discussions on the impossibility to construct a mathematical model of a Multi Agent System,
"But,..., there must be one!!"

Dr. Yuriy Rytzar, for the vulgarization of the watermarking process.
"You don't know what a mean filter is?!?"

Prof. Hans-Heinrich Nägeli, for the lecture, corrections and valuable remarks on the report.
"If JXTA doesn't fit your needs, doesn't that mean that your system is not Peer-to-Peer??"

and Prof. Sviatoslav Voloshynovskiy, *for his extensive knowledge on how to write a scientific paper and for his valuable support to this project.*
"Hum..., only 4 bibliographic references?"

More and more small and middle-range companies are faced to the increasing demand in high performance computing due to overwhelming amount of multimedia content available. Parallel solutions provide a promising workaround for this demand. However, the prohibitive cost of integrated multi-processor systems can block any purchase decisions, reducing the possibilities and sphere of activities of the above-mentioned companies or institutions.

This document proposes an alternative low cost solution. We propose a distributed system of heterogeneous mono-processor stations, load-balanced on Peer-to-Peer basis. The model is based on modified Task-farming approach.

The role of MASTER (or DISPATCHER), is in our model delegated to Peers themselves, therefore reducing the bottle-necking due to transfers of data from Peer to Dispatcher, and then from Dispatcher to another Peer, as it happens in traditional Task-Farming approach. In our study, the Peers manage themselves where to send the surplus of work.

The above-mentioned management is done via message multicast and a strategy based decision-making. We show how to minimize the network as well as processor overhead thanks to this *automated* load-balancing. The resulting prototype demonstrates the ability of our system to support OS-crashes and varying processor load, thus providing a solution particularly effective in an unstable environment. This allows the use of the system on dedicated clusters on one hand, or on the workstations used for text processing and generally low resource consuming administrative work on the other.

The automated recognition of Peers between them facilitates the installation and moreover allows the scalability in a *hot-plug* manner. During the initialization period, the Peer announces itself and integrates automatically in the working cluster. No further configuration is needed.

This report is divided in 9 chapters. The first four chapters introduce the Pervasive image Watermarking and provide some insight on the actual *state-of-art* of the distributed computing. Chapter 5 discusses our Peer-to-Peer model and defines the elements of the network and their roles.

The Chapter 6 discusses some techniques of estimation of the performance of the distributed systems and provides the theoretical model of our Peer-based computing environment.

The last two chapters, 7 and 8, go more deeply in the implementation details of the prototype and summarize the results of performance measures.

The results were concluding and achieve **speedups up to 3.986** with **five** computers which leads to an **efficiency of 79.71%**. During the test phase some of the problems which decrease the performance were spotted. Those, and their solutions are discussed in Chapters 9 and 10.

Contents

1. Introduction

The revolution in IT[1] has brought a demand in parallel computing into the small companies. The increased demand in fast image, audio and video processing has put many of those companies in an embarrassing situation of not having enough computing power.

On the other hand, the personal, middle range computers can be purchased at low cost, which allows to companies, and even households to upgrade their computer systems on regular basis. The lifetime of those computers is in consequence very short. The side effect of this is that a company has, in fact, enough computing power, but it remains unused since those outdated computers are often disencumbered or even worse, thrown to trash.

The company Digital Copyright Technologies SA (DCT) designs and develops applications which allow users to protect their digital data. The main product, everSign, allows the user to digitally sign an image, which grants the same user the legal rights on such an image[7]. The signature is invisible and resistant, independently if the image is stored digitally, eg. on a hard disk, or printed on a physical support, usually paper. The proof of copyright can be provided by using the everSign application and a given cryptographic key.

The signature is added to the image using the Pervasive image watermarking (cf. Chapter 2). As this algorithm uses extensively cryptographic and image processing mechanisms, the demand in memory and CPU is very high, specially when considering very large images, e.g. satellite photographies. It became clear very early that an efficient distributed system will be needed in order to satisfy the company's most demanding customers. The solution had to be scalable in order to address the both CPU and memory limits by adding more computers.

As the watermarking process can be applied to a part of the image, without needing the presence of the whole image in memory, our main idea was to tile the image in a specified amount of smaller images (tiles), and apply the watermarking algorithm to each of them separately. In order to address the scalability issue, the tiles must be able to be processed on an easily upgradeable set of computers. This processing must be load-balanced in order to assure the optimal processing time.

Our system, P-ICE, takes advantage of the obsolete computers allowing their re-use in a cluster, performing the computations in the Single Program - Multiple Data (SPMD) model. Although P-ICE was needed for the particular image watermarking application (everSign Server), the system was meant to be adaptable to other resource consuming applications as well. DCT has engineered and implemented several cryptographic and

[1]IT: Information Technologies

steganographic algorithms which should be distributable with a relative ease by using the same approach.

P-ICE addresses both needs by providing the distributed solution of a concrete application and a consistent framework for distributing similar applications. The only condition a particular data-processing algorithm needs to meet in order to be distributed with P-ICE, is the independent processing of a chunk of data, i.e. the model in which the processing of one set of data doesn't need to interact with processing of other sets.

The computers used to perform the computations execute the particular instance of the P-ICE application. This instance corresponds to the role of that particular computer in the overall process. The networked computer executing a particular instance of the P-ICE application is called a Peer.

2. Pervasive Image Watermark

The watermarking process ([8, 14, 19]) consists in *embedding* a data in the image. This data should be hidden, i.e neither detectable by human eye nor by an algorithmic or mathematical process. This assumes that the sophisticated model of human visual system is used to ensure the perceptual invisibility of hidden data.

This algorithm modifies the color or greyscale values of a limited amount of pixels within an image. The exact positions of modified pixels are computed by a pseudo-number sequence which uses a provided ***encryption key*** as *seed.* The modifications are interpreted as *hidden data* (or ***signature*** in our example). These modifications must be invisible but robust, i.e resistant to well known image processing that could be conditionally divided on filtering (denoising, quantization, blurring, compression, contrast enhancement) and geometrical transformations (scaling, rotation, shearing, change of aspect ratio, cropping and random geometrical distortion)([20, 22]).

Additional error detection/correction algorithms are used to assure the robustness of a watermark. The algorithm is provided as a DLL library.

When we *embed* the signature, we need to provide the algorithm with the image data, a 64-bit key and a 64-bit signature. After the process has finished, the resulting image data has the signature embedded in it. The only way to *retrieve* the signature is knowing the key. When we retrieve the signature we must provide the algorithm with the image data and the key. The algorithm returns the signature (if any) which was embedded in the image using the given key.

The algorithm has the best performance with images with size of 512*512 pixels. The CPU time needed to embed the signature in such an image is given in (B.1).

The memory used by the algorithm when performing the image watermarking can be estimated as follows:

Input image memory allocation:
WIDTH(PIXELS) * HEIGHT(PIXELS) * BYTES PER PIXEL (BYTES)

Output image memory allocation:
WIDTH(PIXELS) * HEIGHT(PIXELS) * BYTES PER PIXEL (BYTES)

Memory allocated for processing:
(F(MAX(WIDTH, HEIGHT)))^2*BYTES PER PIXEL : f(x): next power of 2, starting from x

Sum:
2*(WIDTH(PIXELS) * HEIGHT(PIXELS) * BYTES PER PIXEL (BYTES)) + (F(MAX(WIDTH, HEIGHT)))^2*BYTES PER PIXEL

Examples for a 24bit color image:

Image size	Used memory
512x512	2*(512x512)*3+512^2*3=2.25Mb
1020x1000	2*(1020*1000)*3+1024^2*3=8.83Mb
4096x4096	2*(4096*4096)*3 +4096^2*3=144 Mb
500x4000	2*(500*4000)*3+4096^2*3=59.54Mb

3. Goals

The goal of this project is to construct a distributed environment for embedding and retrieval of signatures in an huge (disk space and network throughput limited) amount of images. We have at our disposal a DLL file which contains the watermarking algorithm. The algorithm provides the methods **embed(key,signature, &image_buffer)** and **detect(key, &signature, image_buffer).** The role of our system is to provide a framework for execution of a big number of instances of the algorithm on different computers at the same time. The requirements defined for this project are:

1. the system must be distributed,
2. efficient load-balancing,
3. scalability,
4. fault tolerant,
5. easily installable and configurable,
6. must perform well in a heterogeneous environment,
7. must be able to handle a huge amount of data,
8. must be adaptable to other, similar algorithms,
9. should use an asynchronous data passing protocol.

The system should have a good performance and be fault tolerant. We want it expandable, easy to install, configure and maintain. When needed, it should be able to interact with a workflow which would provide the images to be signed (eg. e-commerce application). Additional capabilities and decisional algorithms should be integrated with a reasonable amount of time and effort.

4. The State of the Art

The first significant openings in **parallel computing** were made in early 1980s.[21] The first *hypercube* architecture came in 1981 with Caltech's VAX 11/780. From this moment on, the research on parallel computing tend to optimize hardware connections between processors. The main goal was to produce an parallel hardware architecture which is scalable to a very large number of processors. For some topological aspects not discussed here, hypercube was at this moment the most appropriate candidate. Nowadays many topological architectures exist. Every architecture is adapted to a particular set of problems. An example among them, the linear architecture, is appropriate for pipelined parallel computing. More sophisticated crossbar architecture allow modifications of all physical interprocessor connections.

The past decades many efforts were involved in developing hardware but also the programming languages adapted to parallel computing. Major parallel oriented capabilities were added to Fortran (HPF[1]([15])) and C (Parallel C).

With the growing number of networked mono-user/mono-processor computer stations, the idea of **distributed computing** has appeared. Instead of using one multiprocessor computer, the usage of many networked mono-processor stations seemed to be a perfect solution for the low-budget computing. Nowadays we have a large amount of application libraries which allow the parallel code to execute either on dedicated parallel architecture, either on distributed architecture formed of networked mono-processor workstations. We will discuss some of them.

4.1. PVM/MPI

Parallel Virtual Machine (PVM)[4] is an API which allows a heterogeneous network of computers to appear as a single machine. This machine is called Virtual Machine. The development of PVM started in summer 1989 at Oak Ridge National Laboratory. Due to its experimental nature, PVM is available freely.

The goal of MPI (Message Passing Interface)[3] is to develop a standard for writing message passing programs. MPI implements features from a number of existing message passing systems. Started April 1992, MPI was presented in 1993.

Both MPI and PVM provide library interfaces to C and Fortran. These libraries, by their nature, demand the modification of the source code when the architecture of the distributed network changes. As these libraries involve consequent effort to scale, but also to install and maintain, they were not considered as acceptable solution for

[1]HPF: High Performance Fortran

our project. Moreover, by using RPC[2] mechanisms for communication, they are mainly oriented toward synchronous data passing protocols.

4.2. CORBA

CORBA stands for "Common Object Request Broker Architecture"[1]. It was developed by the Object Management Group (OMG). CORBA provides a platform and language independent architecture for writing parallel object-oriented applications. CORBA objects can reside on the same machine or on a dispersed set of computers. CORBA takes advantage of Java object oriented approach.

CORBA wasn't suitable for our model due to it's synchronous data passing mechanism. However, the last version of CORBA implements some of the asynchronous paradigms.

4.3. JXTA

Sun Microsystems introduced a set of libraries aiming to provide all the tools needed to conceive Peer to Peer applications. Project Juxtapose (JXTA)[2], is a set of protocols intended to standardize distributed computing. The JXTA API is written in Java and fits perfectly for communications within a worldwide distributed system. It is optimized and suits very well the Peer-to-Peer communications on the Internet range clusters. The features provided by JXTA would be useful if we decide to apply our model on this kind of WAN's.

The message passing through routers involve additional overhead in order to detect peers, while for obvious reasons, routers do not route the broadcast packets. As our model is simpler due to it's LAN-based architecture, we did not use JXTA. Further work on our algorithm will probably include JXTA in order to enable the communication of many, router separated LAN clusters.

JXTA is an open source project.

[2]RPC: Remote Procedure Call

5. P-ICE

5.1. Introduction

After considering some existing solution for distributed computing (Chapter 4), we have decided to use the Peer-to-Peer approach in order to solve the problem. Our solution follows the actual tendency in abandoning the well known client-server model and giving more liberty in implementation of efficient and robust load-balancing algorithms.

Many other, mainly file-sharing, applications use Peer-to-Peer protocols for data exchange. First SETI then Napster, Gnutella, Kazaa and Morpheus Peer-to-Peer systems provide very popular alternatives for file-sharing applications. Limewire, the well known implementation of Gnutella protocol, in Java, provides load-balanced download of a requested file from several sources. The force of those systems is based on supposition that the information is present at several places, among which one is closer to us than the others.

The needs which led to P-ICE are different. The system which we provide doesn't share files but CPU resources. We could generalize and presume that the CPU's, as the files, are the available resources which we want to obtain in a most efficient way. There is always a processor that has less work to do as the others. The goal is to locate that processor in order to make it perform a job for us.

In nowadays Peer-to-Peer systems there is always a central server to which the Peers connect in order to obtain the list of available resources (or Peers which detain them). This server can go down either for commercial (Napster) reasons, or because of a hardware/software failure. One of advantages of P-ICE is that it doesn't need a central server due to its LAN-based architecture.

The other limit of the existing peer-to-peer applications is that the Peers doesn't have an information on the topology of the resources. When the request for the resource is issued, it is at best (Gnutella) provided to the neighboring peers who provide them further and so on, up to the Peer who can handle it positively. Some times the request for the resource is directly queried from the centralized server (Napster) to which all the Peers connect.

P-ICE handles the information on a resource topology. This information is not queried but communicated periodically by all the Peers too all the others, thanks to network multicast protocols, granted to be effective only in a Local Area Network (LAN).

We have proved trough this document that in mixing two emergent technologies, Peer-to-Peer communications and Multi-Agent Systems, we can provide a highly versatile platform for CPU sharing among individual computers.

5.2. Proposed solution

Our idea was to provide a set of standalone applications which can be executed on a varying amount of personal computers. Those applications should share the same communication protocol in order to communicate and exchange data. The topology of the available computational resources must be known to every running instance of the application in order to address the load-balancing issue. Therefore, this topology is automatically discovered and monitored by the application itself.

P-ICE application allows the computer to behave as a Peer, ie. to provide or receive the different tasks as well as the updated information on the topology of the network. The Peer which participates in a P-ICE network is executing its tasks in daemon mode. This allows the usage of the computer at the same time as a workstation and as a part of the P-ICE cluster. The information about the topology allows the system to load-balance efficiently in such a heterogeneous environment.

The proposed solution encapsulates the image data, signature and key at some starting point of the system, and provide this information as a self-contained task to the cluster of Peers, depending on a discovered topology of the network. The Peers receive this information and execute the algorithm on the provided data. The result is sent to the exit point of the system. The Peers present in the workflow have different roles depending on their location(eg. the Send Peer is located on the image file server and a Scheduler Peer is located on the workstation which performs the initial request for watermarking (entry point for the key and a signature)).

The information on the topology is broadcasted by each Peer to all the others. All the Peers keep the information on the topology of the available resources, and use it to load-balance the tasks. The load-balancing is done individually by every Peer.

The base idea of the solution is shown in figure 5.1, p.16. A scenario of a whole process is described in Figure 5.2 on page 17.

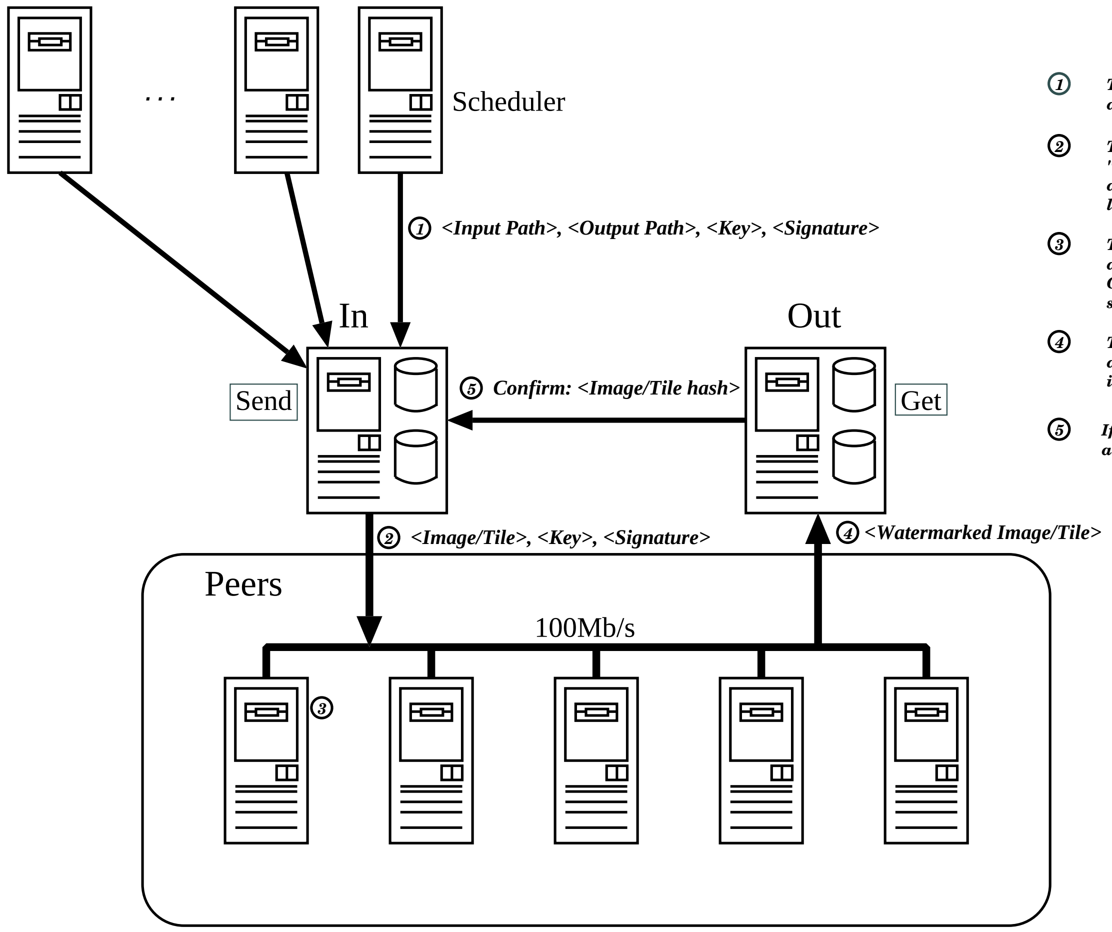
The persistence of the information is assured by the Send Peer in conjunction with the Get Peer.

5.3. Actors involved

5.3.1. Introduction

Some degree of abstraction is needed to understand and use efficiently the Peer-to-Peer architecture. The base actor of our Peer-to-Peer model is a *Peer*. The Peer should be considered as an *entity* composed of the computer algorithm and a set of basic input and output streams.

Within this report we will use some concepts described in [10]. Although the purpose of our work was not to conceive an agent-oriented system, some points are enough similar to be mentioned here. Besides this, the resulting construction is definitely a Peer-to-Peer network and ***not*** an Multi-Agent System (Section 5.3.2). The further work on this system would be to convert it in a Multi-Agent philosophy, which would allow even more efficient load-balancing.


Scheduler
. . .
1 <Input Path>, <Output Path>, <Key>, <Signature>
In
Out
Send
Get
5 Confirm: <Image/Tile hash>
2 <Image/Tile>, <Key>, <Signature>
4 <Watermarked Image/Tile>
Peers
100Mb/s
3

1 *The Scheduler sends Strings containing the paths to images and keys with which the watermark should be done.*

2 *The Send intercepts those Schedules and updates it's "TO DO" list. It loads the images, tiles them if necessary and injects this data in the Peer network, in the load balanced manner.*

3 *The Peers watermark Tiles/Images with provided key and signature, and send the watermarked Data to the Get process. Peers performe a load-balancing based on statistics broadcast.*

4 *The Get receives those Tiles/Images and confirms the arrival to the Send process.Get reassembles the tiles in order to reconstruct the Image.*

5 *If the confirmation didn't arrive within a resonable amount of time, Send resends the Data.*

Figure 5.1.: P-ICE Workflow.

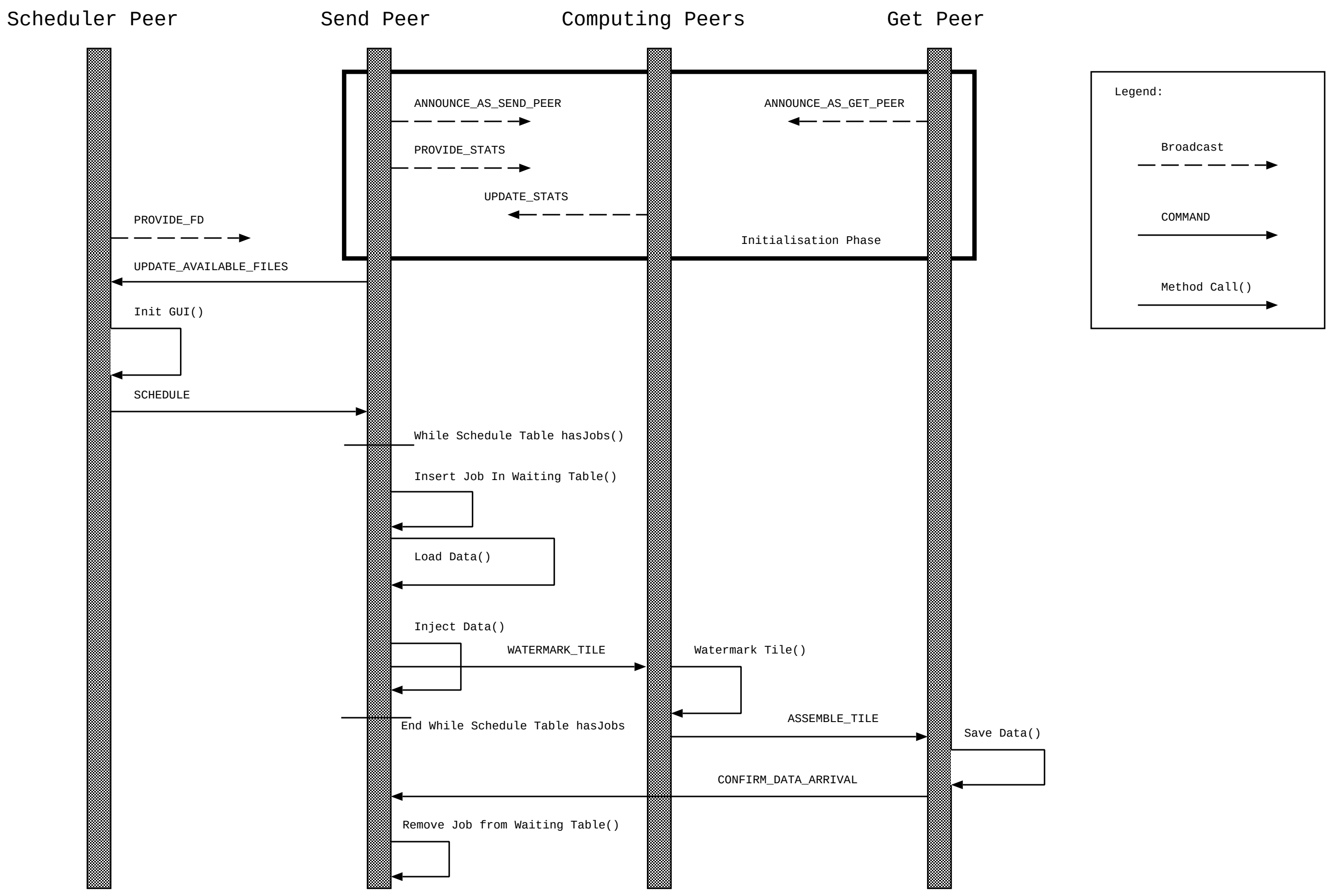


Figure 5.2.: Scenario of the process

5.3.2. Multi-Agent Systems

We should consider briefly the Multi-Agent Systems (MAS) in order to understand the differences and the similitudes between the two models. When we use MAS, we do not consider any of the low-level implementation issues. We consider an agent as an entity with *perceptions, internal states,* and a set of *actions* which the agent can execute. Actions to be executed are returned by a *function* that based on input parameters, i.e. internal states and perception, returns the corresponding action.

There are two main categories of agents. The Agents participating in a MAS can be *tropic,* without memory (the Agent doesn't remember past states or events), or *hysteretic*, with memory (the Agent do remember the past states or events). In the case of hysteretic Agents, the function which returns the action takes one more input parameter which is the past states table.

Our model is constructed in the way to permit its use within a MAS. However, the actions, internal states and the functions are not *formalized* as they would be in a MAS. There is not an unique function which takes the decision on further actions. The decision is brought by a set of *Strategies* which each fit the particular decisional skim. To simplify, we have different functions which control the links between a particular perception and a particular action. To obtain the Multi-Agent System we should unify all those Strategies in one function.

However, we use the Multi-Agent approach in order to schematize the perceptions, actions and strategies. Past events are saved and published in order to obtain the hysteretic model for our Peer.

5.3.3. Definitions

Environment

The *environment* represents the topology of the Peer-to-Peer network. This environment is presented as a set of records on every Peer participating in the network. This environment changes constantly. New Peers can connect to the network, existing Peers can leave the network and the distribution of the load over the network may vary. Some of the Peers may get congested with too many requests, and some other may remain inactive due to lack of tasks. The environment reflects all this changes. The environment is discovered by the Peers via their *Perceptions.*

Perceptions/Stats

The Peer can perceive its environment. The peers that can *perceive* each other (and can communicate with each other) are considered as ***colleagues***. In a further section we will see that not all the peers that participate in the network are colleagues. For example, the Scheduler Peer (Section 5.3.8) is not a colleague of anyone because no Peer can perceive it.

When a peer A *perceives* another peer B, we consider that it has a local copy of data that peer B decides to publish. This data can be published on a regular basis (UDP

Broadcast) or transmitted on request (TCP/IP request, UDP Broadcasted response). We can put about any useful information in this set of data. It can be e.g. the system load, amount of data waiting to be processed or the network load. This data should not be confused with Job data, i.e. the data on which we are performing a specific transformation (Image Data, in our example). That is why we will call it the *Statistics* of a particular Peer. The *Statistics* (*Stats*) are the way one peer *perceives* another Peer or himself (*Internal state* in multi-agent terminology). Some of the pertinent information to publish as *Stats* is listed here:

- List $\{T_1, T_2 ... T_n\} : T_i = Time$ (milliseconds) needed to perform task i
- Number of entries waiting to be processed
- System load
- Network load
- Number of users connected to the computer (if not in a dedicated environment)

Interactions

It shouldn't be important what underlying protocols and network technologies peer *colleagues* use to interact, but the *language* they use. This allows to adapt the model on every communication layer needed. We are supposing that the language level is very simple, composed of exchange of *messages* (for.ex PROVIDE_STATS). The definition of the language used is given in 5.4.2 p.24. Deeper overview of the communication layer used to transport the interactions will be presented in 5.4 p.24. For now we just suppose that the peers can exchange *message objects* containing an arbitrary set of data. Messages are discussed further in 5.4.1 p.24.

Actions

Besides the communications and environment perceptions, the Peers have the ability to perform a limited set of *actions.* Those actions are provided as computer algorithms. [Send Data], [Query Stats] [Embed Watermark] are the perfect examples of Actions. Actions can be triggered by message, or invoked within a particular *Task.*

Tasks

Tasks are specific processes which occur periodically, for ex. at a predefined time interval. The Tasks can serve to monitor some environmental changes, or to execute an action. [Broadcast Stats] is an action triggered mainly by a Task.

Core

The core part of a peer uses the (networked) I/O streams to gather data about its environment, handles requests and performs specific tasks depending on its inputs. The peer can be provided with some basic intelligence. This allows a peer to handle the tasks in a more efficient way. The intelligence of a peer consist mainly in responding to environmental or internal changes with some predefined action. These changes can be detected by the peer's perceptual means, or imposed by a communication request. Our core model is very simple, basically a pre-programmed set of *stimulus-response* paths. The stimulus is represented by the Command*(*5.4.2 p.24*)* received, or environmental change perceived, and the response is the corresponding Action(5.3.3).

5.3.4. Involved Peers

Our model aims to provide a coherent set of colleagues which can perform massive parallel computations. The prototype will be used to perform some mathematical operations, involved in copyright data protection, on a huge amount of images.

The *Peer* is provided with data from another *Peer*. The *Peer* does some computation on the above-mentioned data and returns the results to another *Peer*. The basic idea was that 3 classes of *Peers* are needed. One class that provides the data, one that does the job and the last one which collects the results of that job.

The *Peer* has a perception of its environment. It knows where his colleagues are, and how they are going, i.e. what amount of work they already have. In addition, every[1] *Peer* knows the average time that its colleagues need to do some job (in Stats). Of course, we suppose that *all* chunks of data sent are **of the same size**, therefore we can predict the behavior of a Peer with a certain accuracy; the watermarking algorithm takes the same time to execute on the images of the same size.

The next sections describe in detail all the Peer classes (cf . 5.1, p.16.) Table 5.1 p.21 summarizes the base Peers and their Perceptions, Actions and Tasks.

In order to use P-ICE to distribute a different algorithm, we should modify the actions that Peers perform. Instead of [Embed Watermark] we would have a more generic action [Process data]. However, as P-ICE was developed with pervasive watermarking in mind, the actions are named in consequence.

5.3.5. Send Peer

Send Peer provide the data to other Peers. It consults its *schedule table* in order to know which jobs should be done, in our example which images should be watermarked. When it spots a new (or a next) job, the image data is loaded from disk. The copy of the schedule is written in the *waiting table.* This table is used by the [Check & Reinject Pending Jobs] Task in order to reinject the non-confirmed jobs(Section 5.3.7).

[1]Not all, in fact. Peers that only send the data don't need to announce their statistics.

Table 5.1.: Peer *Perceptions-Actions-Tasks* Card

	Computing Peer	**Send Peer**	**Get Peer**	Scheduler Peer
Perceptions	S(This), L(This), S(CP), L(GP)	S(CP), L(GP)	L(SP)	∅
Actions	[Inject Image] [Send Image To Get] [Embed Watermark] [Retrieve Watermark]	[Load Image] [Inject Image] [Provide FD]	[Confirm Data Arrival] [Save Image]	[Request FD] [Schedule Job]
Tasks	[Broadcast Stats] [Check Load & Distribute] [Discard Pending Peers]	[Discard Pending Peers] [Check & Reinject Pending Jobs]	∅	∅

Legend:
CP: Computing Peer(s) , GP: Get Peer
SP: Send Peer , ScP: Schedule Peer
This: This Peer
S(X): Statistics of X. X∈ [CP,GP,SP,ScP]
L(X): Location of X. X∈ [CP,GP,SP,ScP]
FD: File descriptors

The next step is the choice of the Peer to which the data will be sent. In order to make this choice, Peer provides its *perceptions* to the Distribution Strategy (Section 7.3), which returns the candidate Peer. The data is then sent to that Peer.

This process of load-balanced emission of data will be trough this report called *injection* (Peer *injects* the data into the environment). This injection is made conformable to the particular strategy chosen. As the Send Peer is, first of all, a *Peer*, it has all the information necessary to take this decision (*ie.* all the Stats). The image data is then encapsulated in the Message (Section 5.4.1) along with diverse informations needed for the processing of that image.

The Send Peer knows which data has been sent and it keeps that information until the confirmation of the success has arrived. If that information doesn't arrive within a reasonable amount of time[2], the Send Peer re-injects the data into the Peer Network. This prevents against data loss due to OS or hardware malfunction. *Unfortunately*, if the same happens on Send and Get Peers, there will be no one to resume the job. For this reason this two services are the weak part of the system and should be protected in an adequate manner. For now, this situation is an acceptable compromise if we consider that there is lot more Computing Peers than Send and Get Peers (about 1 to 10 should be a reasonable bet), so the majority is well protected. More, Send and Get Peers can run on about any UNIX environment which supports the JAVA Virtual Machine (JRE1.3_1).

The Send Peer doesn't broadcast it's statistics while it doesn't receive any data except the confirmation of success of the operation. That reduces the network overload on the Send Peer (6.2,p. 26). Details on implementation of the main Send Peer threads are given in Section 7.5.4.

5.3.6. Computing Peer

Computing Peer is the main brick of our system. Its job is to embed or retrieve a signature to/from an image. It receives the messages which encapsulate all the data that a specific image processing algorithm needs. When the embedded algorithm returns the result (ex. watermarked image), Peer forward the data to the perceived destination (**L(GP)**). It periodically checks it's load (perceive it's Stats) and, if needed (based on a particular Strategy), injects the surplus of data to it's most available colleagues (most available according to a particular Strategy). This is achieved trough *Check Load & Distribute* task. The Action which is executed by this Task is [Inject Tile]. Further information concerning Strategies is provided at 7.3,p. 34.

Computing Peer is a daemon. It is executed once and it waits on it's inputs for the commands.

5.3.7. Get Peer

The Get service is provided by the Get Peer. During the initialization phase, *Get* announces itself as a *receiver* for the resulting data. From this moment all the concerned

[2]Some kind of average *traversal time.*

peers perceive **L(GP)**. The Computing Peers put their results in a buffer until this announce has been made. After this, all the results are flushed to the Get Peer. Get Peer saves the received data on a hard disk.

Besides this, the Get Peer has the responsibility of post-processing the data. In our case it reassembles the tiles in one image. Moreover, the Get Peer confirms a successful arrival of the data to the Send Peer (Action [Confirm Data Arrival]). Details on implementation of the main Get Peer threads are given in Section 7.5.5.

If the Get Peer crashes, the system stops. Even worse things that happen if this occurs, along with the solutions to this problem are discussed in Section A.2.1.

Get Peer is a daemon. It is executed once and it waits on it's inputs for the commands.

5.3.8. Scheduler Peer

Our system is envisaged for usage with high data throughput workflows. The quantity of data to process is tremendous, virtually infinite. That's why we consider the data as the flow and not as the quantity. If Send Peer is executed on the predefined amount of data (for example process of the whole directory of images), it must wait until Get Peer confirms the arrival of **all** the data before quitting. For injecting further data into the Peer Network, we must re-launch the Send Peer.

This is not very efficient when working with data flows. The time wasted waiting for confirmations can be used to inject more data. For this reason Send Peer works as a daemon. It waits on his inputs a message indicating what data to send into the network. At the same time, one of his tasks is to re-inject the data for which it didn't receive a confirmation of successful arrival.

Scheduler Peer is launched by a user (or any Internet technology based script). Upon the initialization phase, the request for file descriptors [Request FD] is broadcasted. Send Peer replies with it's file descriptors. Those are returned to the user via the GUI. The user fills the key and the signature fields, chooses which files and/or directories to process, and clicks on a desired action in order to send the job. The Scheduler Peer then broadcasts the request into the environment. Send Peer perceives that request and inserts it in his schedule table.

Scheduler Peer is not a daemon. Although when the jobs are scheduled it doesn't quit, which allows the schedule of further jobs, the received Send Peer file system descriptors are not updated. It is not recommendable to update those periodically because of Send Peer performance concerns. Some issues on this are discussed in Section A.2.2.

Initially, the Scheduler Peer doesn't know where the Send Peer is, while the latter doesn't publish it's Stats. That is why it broadcasts it's requests. Send Peer however reply to the Scheduler Peer directly, because it responds to a message. The good secondary effect is that from now on Scheduler can schedule jobs without "disturbing" the Send Peer (It conserves the network address of Send Peer).

Table 5.2.: Used Commands

Command name	Sender	Receiver	Data Type	Socket	Description
WM_IMAGE	Send	Peer	Tile	Data	be watermarked!
WM_TILE	Peer	Peer	Tile	Data	be watermarked!
UPDATE_STATS	Peer	Peers, Send	Peer	Multicast	Here are my stats, update locally!
PROVIDE_STATS	Peers, Send	Peers	Peer	Multicast, Control	Give me your stats!
ASSEMBLE_TILE (If applicable)	Peer	Get	Tile	Data	This tile is watermarked, assemble the image!
ANNOUNCE_AS_GET_PROCESS	Get	Peers	Peer IP	Multicast	I announce as get process.
ANNOUNCE_AS_SEND_PROCESS	Send	Get	Peer IP	Multicast	I announce as send process.
SCHEDULE	Scheduler	Send	Scheduler Entry	Multicast	Do this.
QUERY_SEND	Scheduler	Send	Peer IP	Multicast	Send Peers, please provide your available file descriptors.
UPDATE_AVAILABLE_FILES	Send	Scheduler	Filesystem descriptor	Control	Here are my available files.
CONFIRM_DATA_ARRIVAL	Get	Send	Hash code	Multicast	Data has successfully arrived.

5.4. Interactions

5.4.1. Messages

Messages embed the Commands(Section 5.4.2) and data that Peers want to communicate. The Messages can be sent to the specific Peer or broadcasted. A Message is usually issued by an Action or a Task.

5.4.2. Commands

Commands are character string data encapsulated in a message sent by the Peer. Table 5.2 on page 24 shows the commands used in our model. For the structure of the message object please consult Figure 7.1 on page 31.

6. Theoretical performance

6.1. Performance evaluation methods

Within this chapter we will remind some techniques used to express and compute the efficiency of a distributed algorithm. For further lectures please consult [17, 11].

Speedup

Given a single problem with a sequential algorithm running on one processor and a concurrent algorithm running on **p** independent processors, Relative Speedup is defined as:

$$S_p = \frac{T_{sequential}}{T_{parallel}}, \text{ p = number of processors.}$$

Relative Efficiency

Relative Efficiency is defined as:

$$E_p = \frac{S_p}{p}$$

It computes what percentage of available computing resources is used for performing the job. 1-E_p is the overhead due to additional tasks involved in data distribution and transfers.

Amdahl's Law

Amdahl's Law ([5]) expresses the speedup gained using multiple processors versus single processor, for accomplishing the same task. The ***speedup*** for a given program is defined as:

$$S = \frac{SeqP+ParP}{SeqP+\frac{ParP}{p}} = \frac{1}{SeqP+\frac{ParP}{p}}$$

SeqP= portion of the sequential part of the algorithm, ParP= portion of the parallel part of the algorithm, p=number of processors

Amdahl's law defines the ***maximum speedup*** achievable by a parallel computer with p processors as follows:

$$maxS \leq \frac{1}{\alpha + \frac{1-\alpha}{p}} \leq \frac{1}{\alpha}, \alpha = \frac{SeqP}{ParP}$$

For additional information please consult [11, 16, 17].

6.2. Theoretical performance of our model

Amdahl's law clearly states the limit of theoretical speedup of an algorithm. To simplify, a parallel algorithm with 10% of it's work done sequentially cannot achieve better speedup as 10, regardless of the number of processors used to solve the problem.

In our model this is applied to Send Peer. If the ratio between parallel portion of the algorithm ([Embed Signature] for ex.), and the sequential portion of the same algorithm ([Inject Data]) is 0.1, the maximum speedup would be 10. For achieving this value we should have at least 10 processors. For some bound effect concerns the ideal number would be 11 (consult the figures 6.1, 6.2 and 6.3). If we use more Peers than that, all the additional computing resources won't be used. If we send 10 images in time needed to watermark one, the eleventh Peer will receive the data when first Peer finishes his job. On the other hand, the 12th Peer will receive data while first Peer is ***waiting***. The first Peer will lose 1/10 of potential computing time waiting to receive the new data. As network speed is supposed constant, we adapt the number of Peers (11 in this example) with above mentioned concerns in mind.

The goal of P-ICE is to provide a platform which allows experimentation of different load-balancing techniques which would *approach as much as possible* the theoretical performance. The overhead due to data transfer is supposed non modifiable. The hardware solutions or better data compression would be needed, but due to low-cost concerns are not proposed/discussed here. Concepts discussed in 5.3.3, p.18 are used to grant the *availableness of data* for each Peer at every moment of the process and hopefully with a computational overhead as low as possible.

Figures 6.1 , 6.2 and 6.3 show the delays caused by the inaccurate number of Peers. The black data processing bars on the figures include the non-network overhead. Please note that the peer order (horizontally from left to right) on this figures is such only for esthethical reasons. The order in which the incoming data is sent to Computing Peers is defined by a Strategy. It can be cyclic (P_1,P_2,P_3,P_1,P_2,P_3,...), random (P_2,P_1,P_3,P_2,P_2,P_1,...), ..., or dynamically generated, as in our model. The figures show the situation to which we tend. Every Peer, when his job finished, must have the data ready for further processing.

This is achieved through different Distribution Strategies (7.3) among which at least one works better than the others. The aim of this work was not to find that particular strategy, but to construct the working model which provides enough flexibility and robustness for being applied on the modern or old hardware. However, some strategies are discussed in further chapters.

Figure 6.1.: Optimal # of Peers

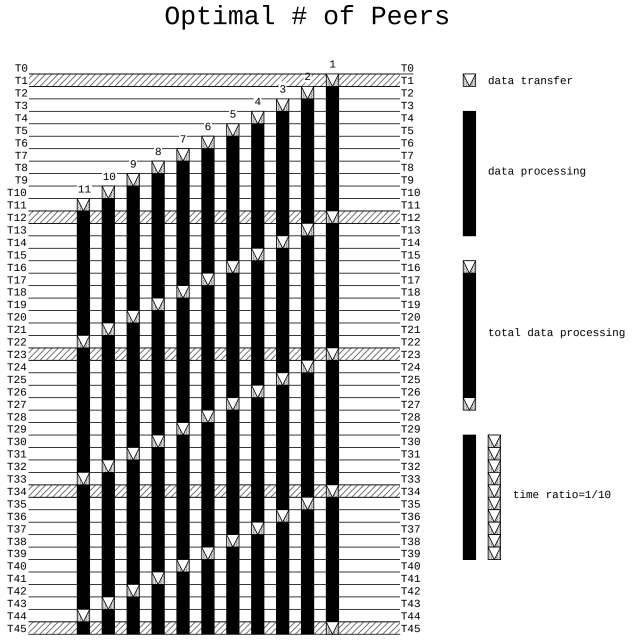


Figure 6.2.: Supra-optimal # of Peers

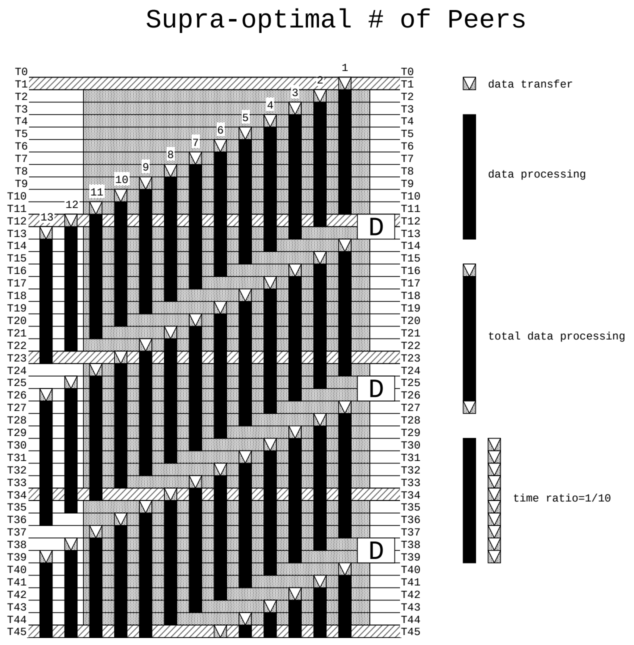
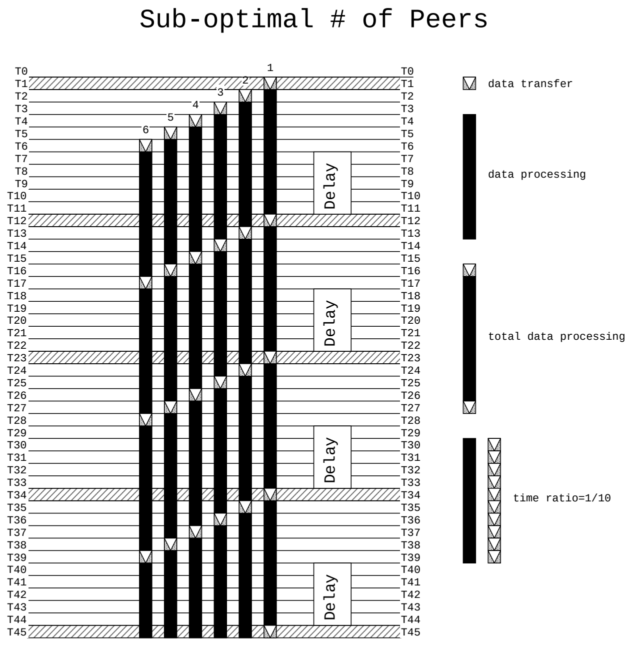


Figure 6.3.: Sub-optimal # of Peers



In order to assure the optimal performance of our system we define some rules to be respected:

1. Send Peer doesn't wait. To satisfy this requirement we use the needed number of Peers. In addition we should assure that the Get Peer saves the data as fast as the Send Peer can load it.

2. Every Peer has always an available task to execute. This is achieved trough well implemented Distribution Strategies.

3. The network overhead due to Peer load balancing provoked by the overloaded buffers [Check Load & Distribute], should be minimized. This is achieved trough the solutions of points 1 and 2.

6.3. Deadlock avoidance/prevention

Deadlock is a state of the system in which processes are blocked, waiting on an event that only other (actually blocked) processes can provide. There are four condition which may lead, ***if all met***, to the deadlock of the system ([6]):

1. **Mutual exclusion** (mutex): only one process can access a resource at a given time.

2. **Hold and wait**: a process demands a resource while already holding one.
3. **No preemption**: the resource cannot be forcibly taken from the process.
4. **Circular wait:** circular graph in which all the processes are waiting the release of the resource by the next process, in order to continue.

Deadlock prevention consists in the assurance that the mentioned conditions for the deadlock are never met ([13]). *Deadlock avoidance* applies on systems which cannot negate these conditions. Deadlock avoidance imply the usage of different algorithms (eg.Banker's algorithm) in order to detect the future states of the system in which the deadlock can arise ([9]). Those states are then *avoided* with different resource allocation.

In our system the deadlock cannot happen as there is always a loophole for the data evacuation (no circular wait). Even if three or more processes want to send data circularly to each other, and all those processes have the buffers full, we have the guarantee of deadlock resolution because of the Get Peer, to which all the peers send the processed job data, which empties the buffers. Of course, if on get Peer there is no more disk space, we will experience a halt of a whole system.

7. Prototype implementation

7.1. Analysis

Figure 7.1 on page 31 shows the architecture of a Computing Peer. The Computing Peer consists of two main parts: The **Watermarker** and the **Dispatcher**. While the Watermarker limits itself on processing the image data, the dispatcher assures the availability and recuperation of data. Following subsections give some more insight on those two parts.

7.1.1. Dispatcher

Dispatcher is the *"head"* of the Peer. It handles the network requests and performs specific maintenance tasks:

- **Message processing**: Commands are extracted from messages and executed.
- **Load-balancing**: The surplus of job to be done is extracted from watermarker input buffers and sent to less overloaded Peers.
- **Sending and retrieving of data from the watermarker**: the data is inserted in watermarker's input buffer in a controlled manner in order to avoid buffer overloads. When the watermarker output buffer receives data, the dispatcher is notified in order to proceed to the evacuation of that data.

All the Peers have a dispatcher. The behaviour of the dispatcher depends however on a type of service this Peer provides (Send, Get, Schedule or Compute). Although the core procedures are the same for all types of peers, the reaction on incoming messages depends of type of service. For example, the Get and Schedule peers don't reply on PROVIDE_STATS broadcasts.

7.1.2. Watermarker

The Watermarker object is a java thread with an input and an output buffer. Watermarker thread retrieves data from the input buffer, performs an operation on that data and inserts the processed data in the output buffer. The routines for embedding and detecting watermark are provided as Windows DLL's and therefore are accessed via the Java Native Interface (JNI)[18].

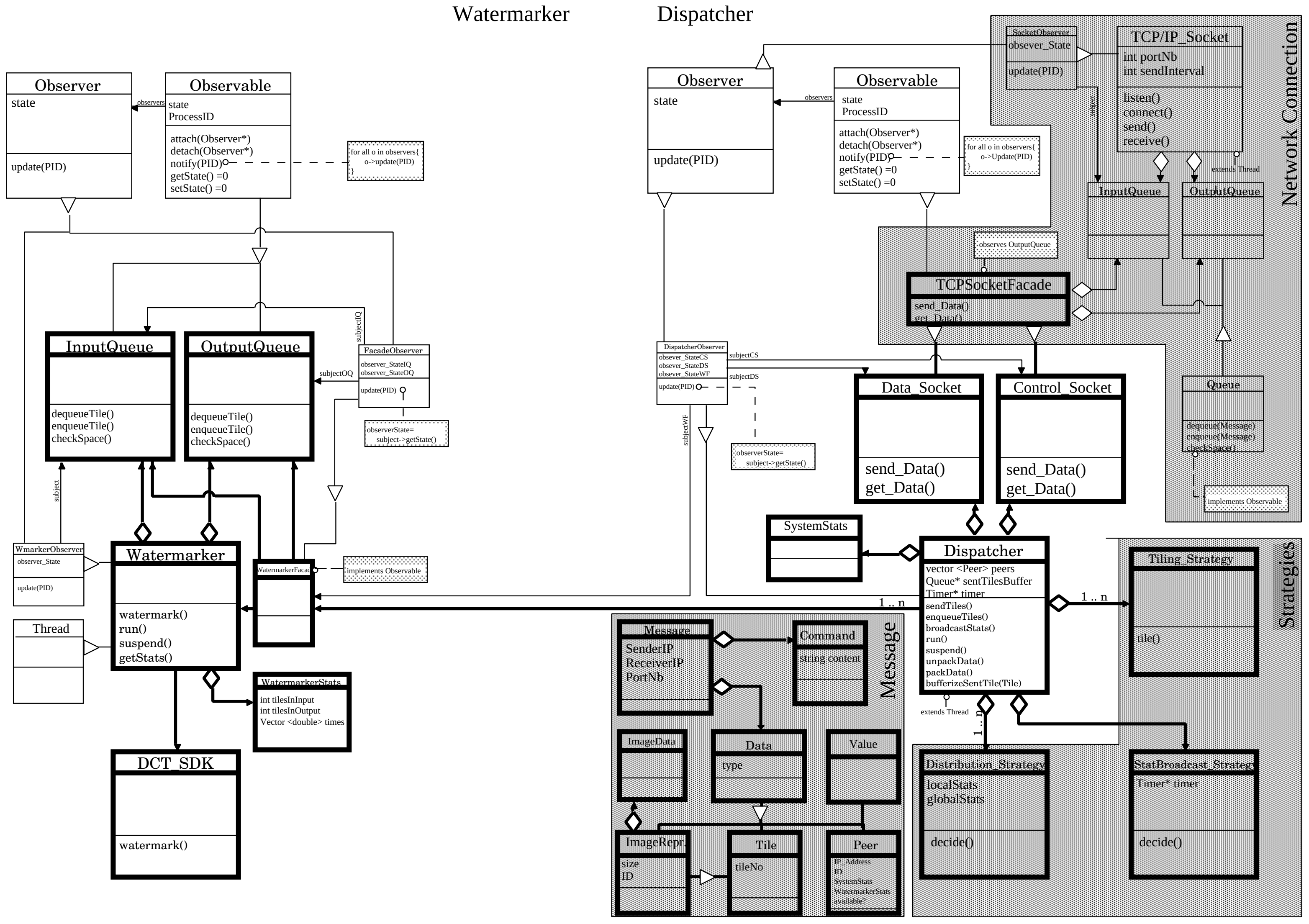


Figure 7.1.: Peer Architecture

Some image manipulation is needed to convert data from it's compressed form (JPEG, PNG) into the raw RGB model needed for the Watermark embedding and retrieving DLL's. The DLL accepts only the byte array with aligned R, G and B (8-bit) components of each pixel. The needed conversions are done using *java.awt.image* package. The prototype application doesn't support the tremendous amount of image codec's present in Java Advanced Imaging (JAI) libraries. As the JAI is an add-on on Java SDK, we don't use it in the prototype application. One should, in order to support more image types, install the JAI. However, JPG and PNG image formats are supported without the JAI.

Only the Computing Peer has the Watermarker object. The Watermarker main procedure checks continuously the presence of data in the input buffer. When this data becomes available, it is extracted, processed and inserted into the output buffer. The output buffer notifies the dispatcher via the Observer design pattern[12]. The dispatcher then extracts the data and sends it to the Get Peer.

7.1.3. Message

For transporting the messages we are using a standard Ethernet switched or hubbed network with broadcast capabilities. The lower layers are traditional TCP/IP or UDP layers. Java language adds a tremendous set of APIs for network socket programming. They are provided within Sun's JDK (Java Development Kit). All the necessary information about Java Socket API is available at "http://java.sun.com". The message object structure is shown in Figure 7.1 on page 31. Each message consist of the command string and an attached data object. The receiver (unless it is a broadcast) and the sender fields are also provided. The message object is serialized and piped through socket to its destination.

Network

We use Java's SocketAPI to perform network communications. These communications are socket oriented. We are not using any middleware for performing those data transfers in order to make our model portable and platform independent. Java VM exist for many hardware architectures.

Unicast

Unicast transfers are used for sending job data. The integrity and packet order are regulated by the TCP/IP stack. We don't add additional control structures. In case of brutal malfunction of a receiving peer, the sender is notified via an java exception which is properly handled.

Multicast/Broadcast

Multicasts/Broadcasts are used for publishing statistics on the Peer network. Broadcast Messages are piped to Java's Multicast/UDP-based socket. As the UDP protocol doesn't handle the order of packet arrival, we send only one packet at each broadcast. The size of this packet is limited by the predefined receive buffer size. It could be dynamically allocated, but this would need additional messages to be exchanged. As our goal is to minimize broadcasts, this was not an acceptable solution. Each Peer must check the size of the packet before broadcasting.

Routing hardware

For the technical reasons we suppose that the Peer network resides within the same LAN. As multicast packets are usually rejected by routers, technologies like SOAP or JXTA should be integrated in order to perform WAN or Internet broadcasting. Peers can be connected by the coaxial cable or hub (RJ45), but performance is better if we use network switches (collision avoidance).

7.1.4. Sockets

First of all, please note that the socket we are talking about is a particular socket which implements the Peer-to-Peer paradigm. In Peer2Peer communication each participant has a **client** and a **server** network socket. That's why, from the networking point of view, our P2P socket is composed of two generic network sockets. When we use the term "socket" we refer to the Peer-to-Peer socket.

There are three different socket types:

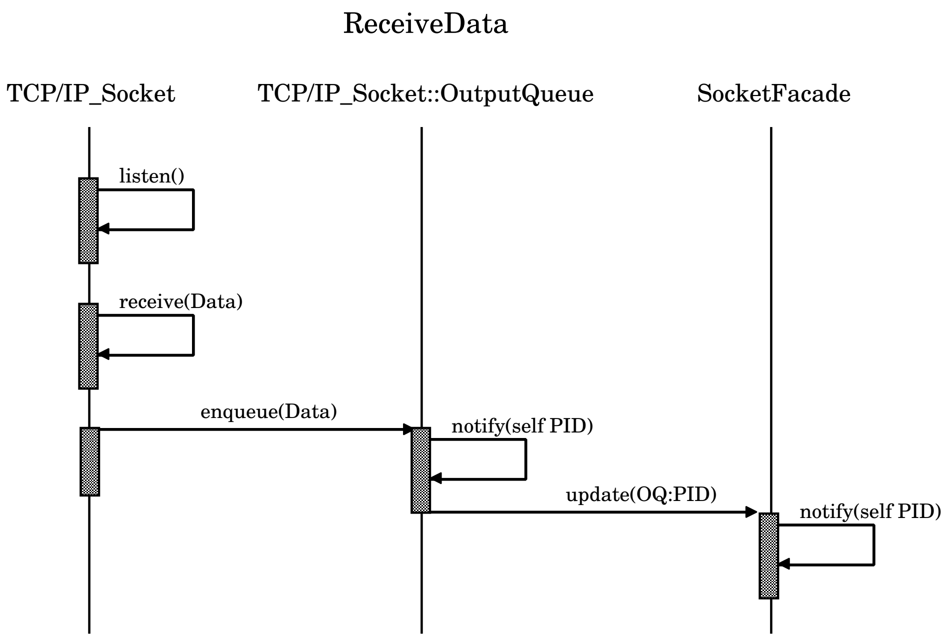
1. **Data Socket**: for sending and receiving job data (Images).
2. **Control Socket**: for sending and receiving control data (available files).
3. **Multicast Socket**: for broadcasting of stats and announcements of available Peers.

The sockets are based on *java.net* package. In addition, two buffers hold the incoming and outgoing messages of each socket. The buffers(Section 7.5.1) are present because of the asynchronous message passing model. They compensate the differences of processing time between Peers.

The socket is composed of the server and the client part. The server part receives incoming connections. When a connection is received, the socket verifies if there is enough space in the buffer for the next message. If it isn't the case, the server socket is not opened until the space frees.

The client part of the socket is notified by the input buffer (see Section 7.5.1) . It then extracts data from the input buffer and sends it. If, for some reason, the data can not be sent to the destination, the socket retries several times, and then drops the message and continues. We rely on the overlying protocol between Send and Get peer (CONFIRM_ARRIVAL) in detecting the dropped jobs. Those jobs are reinjected in the network if not confirmed as done (Section 5.3.5, Section 5.3.7).

Figure 7.2.: Interactions while receiving data



7.2. Internal data exchange Protocols

Peer's internal data exchange relies firmly on *Observer-Observable* design patterns. Figures 7.2 to 7.6 show the sequence diagrams of Peer internal protocols.

7.3. Distribution Strategies

Distribution strategies are used to load-balance the data between Peers. Send Peer uses those strategies in order to decide to which Peer to inject the data. Computing Peers use the same strategies to discharge some of their data in order to avoid overloads. It is not necessary that all the peers use the same strategy.

Although the goal of this project was not to provide any particular load-balancing strategy, for the testing purposes we have developed several, very simple algorithms. Those algorithms provide essentially decisional patterns which based on environmental perceptions decide to which peer to send data. Different distribution strategies are presented below:

- **First Peer Strategy**: The data is sent to the first Peer in the list.
- **Random Peer Strategy**: The data is sent to the randomly chosen Peer.
- **Circular Peer Strategy**: The data is sent to Peers in the order they appear in the list.

Figure 7.3.: Interactions while sending data

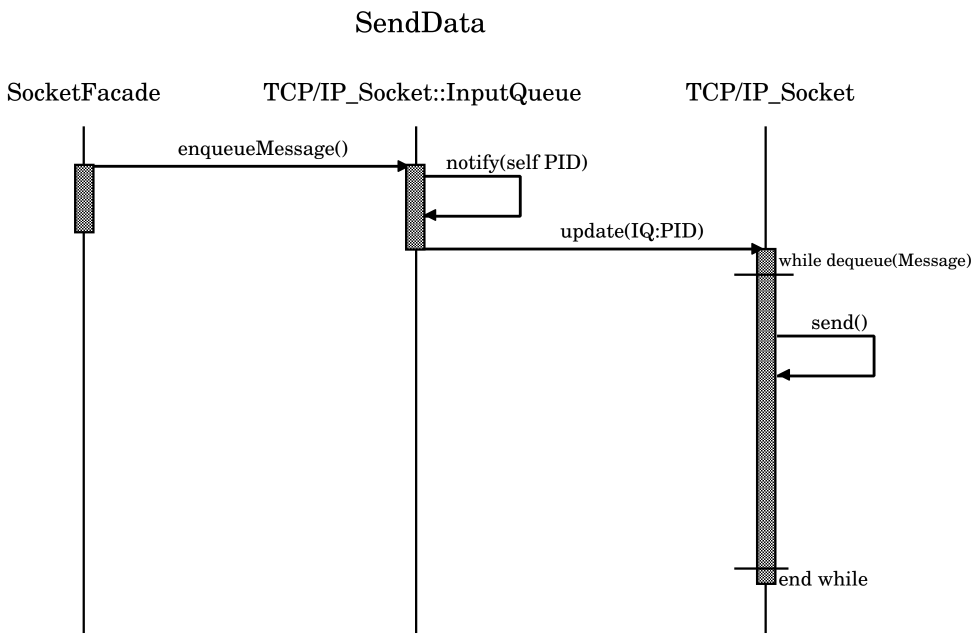


Figure 7.4.: Interactions while receiving tiles

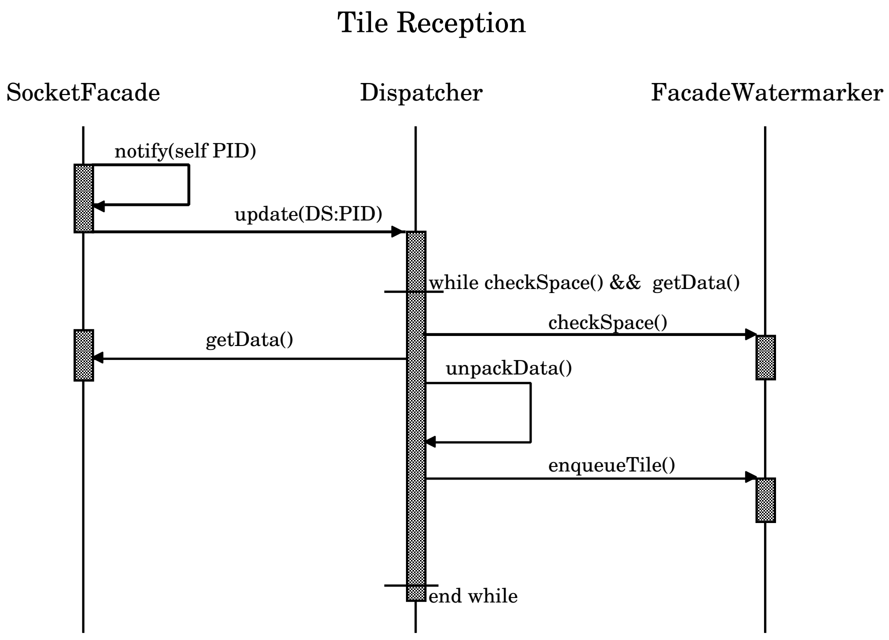


Figure 7.5.: Interactions while sending tiles

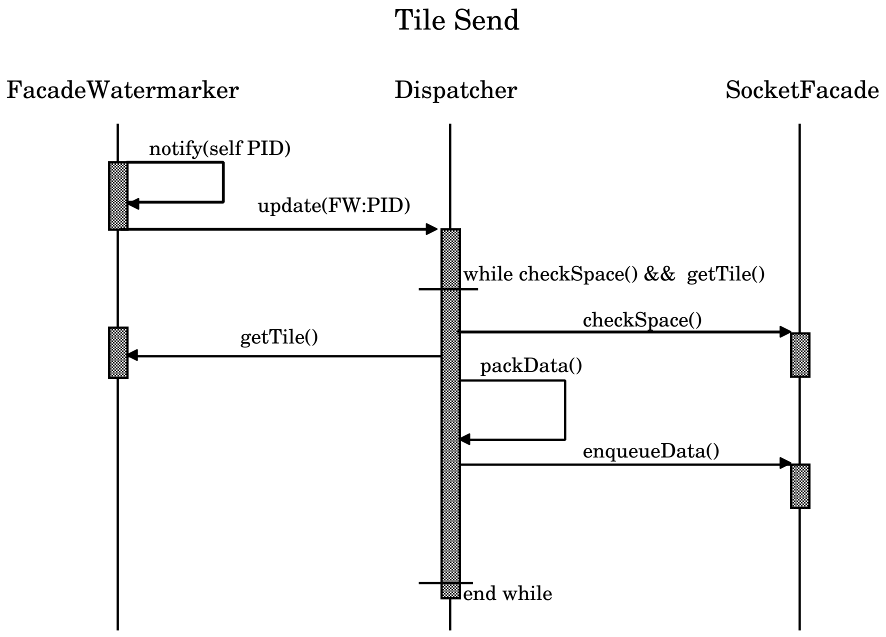
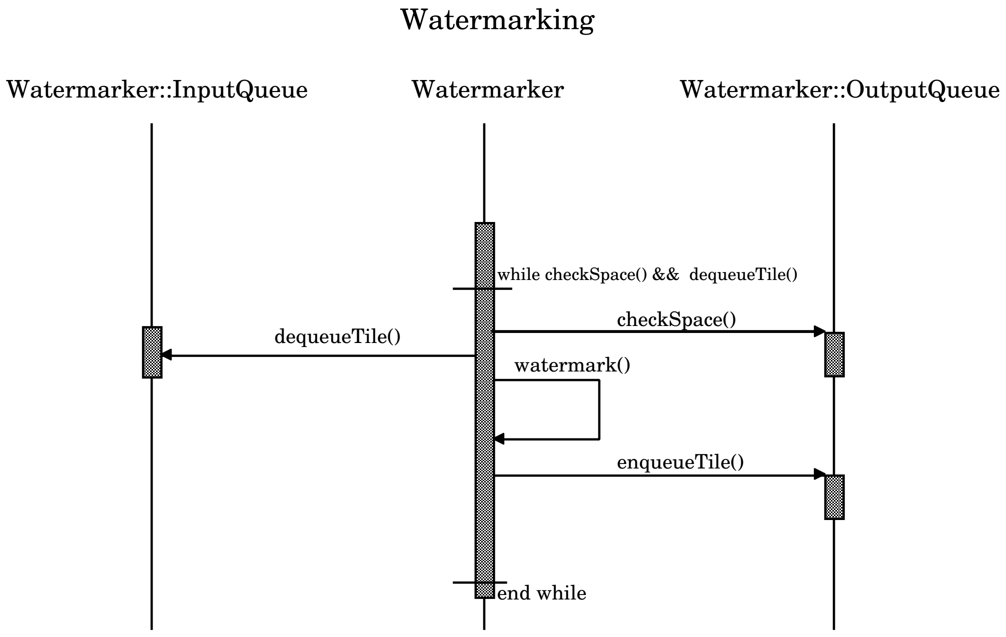


Figure 7.6.: Interactions while Watermarking



- **Smallest Input Queue Strategy**: The data is sent to the Peer that has the "emptiest" Watermarker input queue (buffer).

As envisaged, the strategy that gives the best data distribution is the Smallest Input Queue Strategy. This list is not exhaustive and should be updated when further work on the application is done.

7.4. Broadcast Strategies

Broadcast strategies regulate the publishing of a Peer's Stats. This type of strategies is used by the BroadcastStats Action(Table 5.1 on page 21). When the Broadcast Stats Task decides to trig the Broadcast Stats Action, the Action consults its Strategy, which returns if or if not to publish the stats. In our implementation, the only implemented Banal Broadcast Strategy returns always true, which implies the fixed interval broadcasts, which however can be initialized at a desired value via the Task period.

In further work it should be good to base the decision to publish or not on number of available Peers. This is further discussed in Section A.1.2.

7.5. Important algorithms

7.5.1. Buffers

The most "tricky" part of our system is the network data exchange and buffering. As our model is, as stated above, an asynchronous model, the buffers are the quite important part of the system. The buffers are used at the socket part (Section 7.1.4) and a Watermarker part (Section 7.1.2) of the Peer. The Watermark buffer compensates the varying speed of the data processing algorithm, while the Socket buffers compensate the variations in network load.

Those buffers are based on a linked list which acts as a queue. The queue implements the *Observable* interface in order to notify the observers on its status change. This is used in order to trig data processing from the Socket output queue and sending of the data from the Watermarker's output queue to the Get Peer.

7.5.2. Message processing

When the Dispatcher is notified upon a Message arrival (by the Socket output buffer), it calls the processMessage() method. This method extracts the Message from the buffer, and de-capsulates the Command. Depending on the Command and the Peer type (Computing, Get, Send or Scheduler) , the particular action is taken. If necessary, the data is extracted from the Message.

7.5.3. Initialization phase

During the initialization phase, the Peer creates all the necessary objects (Sockets, Watermarker, ...). Then, it queries its IP Address in order to set the sender field on its future messages. The Sockets are opened and the Stats broadcast performed. Peer then, depending on its role, executes the corresponding main thread.

7.5.4. Send Peer Main Thread

The following java algorithm is executed on the Send Peer:

```
while(true){
   while (scheduleTable.hasEntries()){
       SchedulerEntry job=scheduleTable.getNextWaitingEntry();
       waitingTable.insert(job);
       Tile tile=new Tile();
       tile.loadFile(job.filename());
       tile.setKey(job.getKey());
       tile.setSignature(job.getSignature());
       myDispatcher.injectTile(tile);
   }
   yield(); //avoiding 100% CPU load
}
```

All the other services are provided by Tasks and Commands.

7.5.5. Get Peer Main Thread

The Get Peer has no main Thread because of its passive nature. On initialization it broadcasts its IP, via the ANNOUNCE_AS_GET command. The service for recuperation of processed data is provided via the ASSEMBLE_TILE Command, which is issued by Computing Peers. There is however an implicit child Thread, Server Socket, which runs as daemon and keeps the Peer running.

7.5.6. Computing Peer Main Thread

The Computing Peer is also an passive Peer. It does nothing by himself but its Tasks. Those Tasks, with Sockets and Watermarker are however active as separate Threads.

8. Results

8.1. Performance

Our model was tested in order to provide a "proof-of-concept" of the underlying technique of distributed computing. The goal of the test set is to allow the estimation of the speedup and efficiency (see Section 6.1) of our algorithm. The test cluster is composed of seven peers: Send Peer, Get Peer and five Computing Peers. The hardware is described in Table 8.1 on page 39.

The test job is composed of 560 tiles with dimensions of 512 by 512 pixels. This value represents the optimal dimension for the Watermark embedding algorithm. The tiles were obtained by cutting 4 huge images using the software developed for the occasion. The software is based on JAI (Java Advanced Imaging) Libraries. The measured speedup is showed in Figure 8.1 on page 40 , and the efficiency in Figure 8.2 on page 40 .

The serial processing time (Section 6.1) which we use to compute the efficiency of a set of Peers is obtained by averaging the serial processing times of the concerned Peers. Serial performance results are showed in Section B.1, and parallel performance measures in Section B.2.

The serial measures were obtained using a Serial Peer, coded for the occasion, which loads files, watermarks them and saves them to the disk. The processing time is obtained by the subtraction of the system time at the end of the process, from the time at the beginning of the process.

The parallel measures were obtained using a stopwatch. Processing time is measured from the moment the user schedules the jobs using the GUI, to the moment the last job

Table 8.1.: Test cluster hardware

Name	Role	OS	CPU Model	CPU Clock	RAM	Network
Peer1	Computing	NT 4 Server	Intel Pentium II	346 Mhz	256Mb	10 Mb/s
Peer2	Get	NT 4	Intel Pentium II	345 Mhz	256Mb	100Mb/s
Peer3	Send	Windows 2000	Intel Pentium II	394 Mhz	256Mb	100Mb/s
Peer4	Computing	NT 4	Intel Pentium III	544 Mhz	256Mb	100Mb/s
Peer5	Computing	Windows 2000	Intel Pentium III	544 Mhz	128Mb	100Mb/s
Peer6	Computing	Windows 2000	Intel Pentium III	544 Mhz	128Mb	100Mb/s
Peer7	Computing	Windows 2000	Intel Pentium III	725 Mhz	256Mb	100Mb/s

Figure 8.1.: Speedup

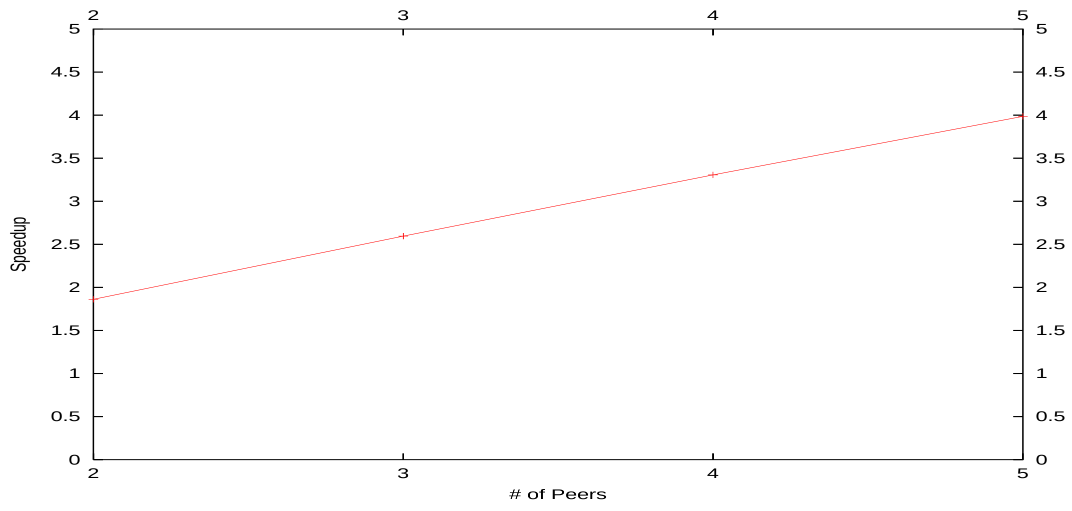
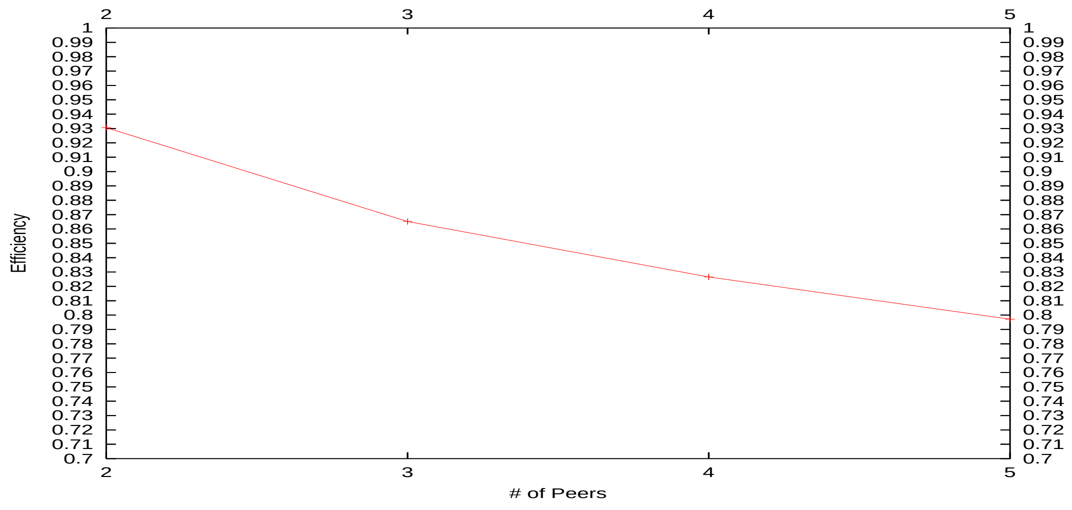


Figure 8.2.: Efficiency



arrives to the Get Peer.

The result show that the speedup behaves linearly, which is a good thing. Efficiency tend to diminish with increasing number of Peers, but we can notice that the curve stabilizes near the end. It would be interesting to test the system on a greater set of computers.

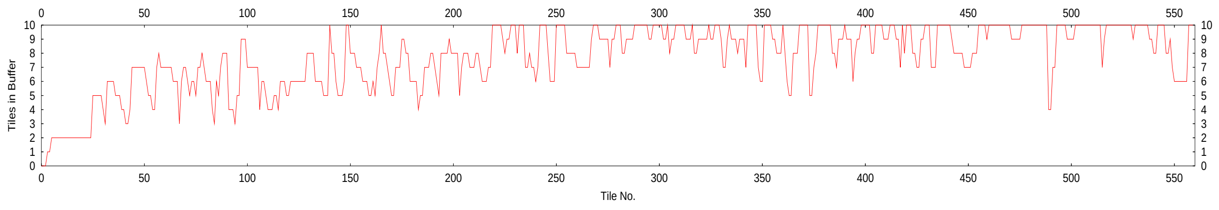
As the Send and Get Peers reside on file servers and do not participate in the computing, they were not counted during the efficiency calculus.

8.2. Data distribution

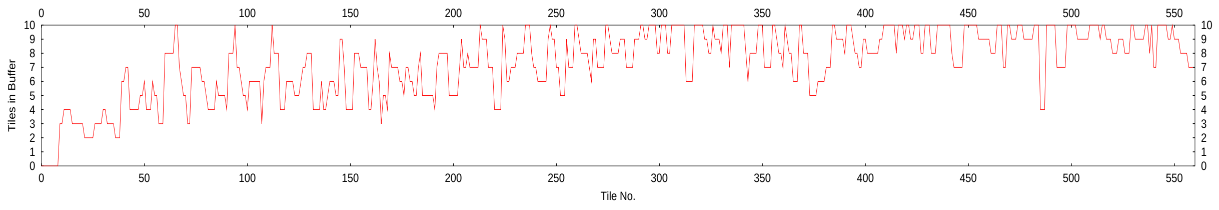
In order to confirm the needed condition described in Section 6.2 (every Peer works), we have produced the charts presented in Figure 8.3 on page 42 . Those charts show the states of the input buffers of the Peers at the moment the Send Peer chooses to inject the next job. The measures were obtained using a slightly modified SmallestInputQueue Distribution Strategy, which, when called in order to provide the Peer to whom the job is to be injected, saves the perceptions (Computing Peers input buffer status) in a text file.

The important thing to notice is that there is **never** a Peer without a job. When some Peer has its buffer almost empty, the load-balancing algorithm (Send Peer or other Computing Peers) fills it very rapidly.

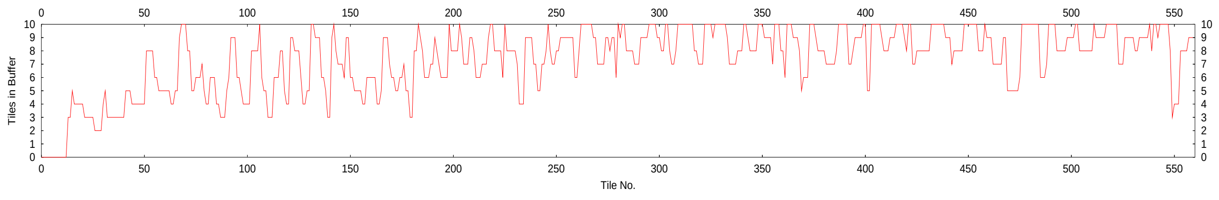
Figure 8.3.: Distribution of Data in the Peer network (Peers 1, 4, 5, 6 and 7)



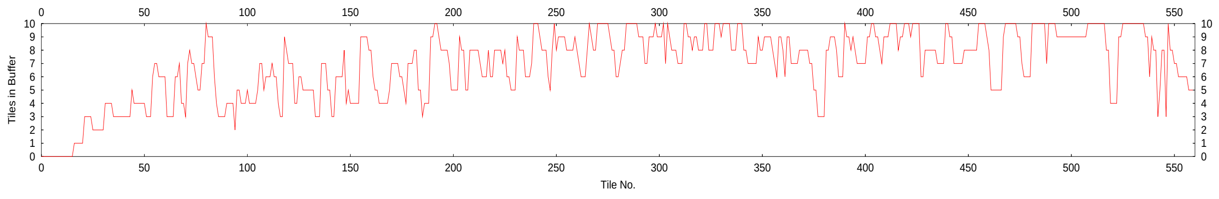
(a)



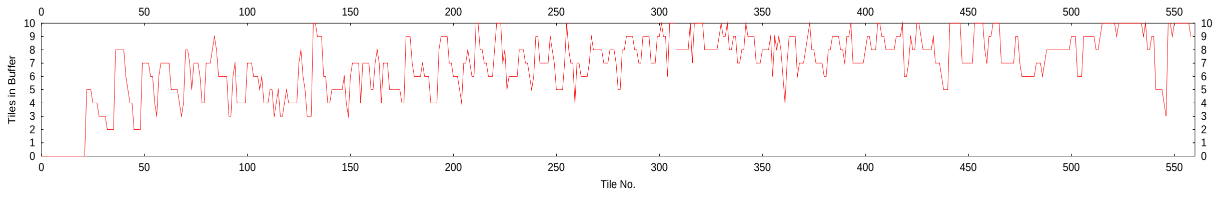
(b)



(c)



(d)



(e)

9. The future of P-ICE

The future of P-ICE and its possible successors reside in huge, very fast, WAN based networks. Today, we can purchase copper-based Gigabit network hardware at less than 36$, providing 600Mbit/s throughput, opening a whole new field of computer technologies and applications.

In the future, modern offices, all the hardware will participate in performing the tasks at its own level. Pocket PC's, electronic version of our agendas, already provide Firewire interfaces for high rate data transfers. Why not using this high performant computers (eg. Intel StrongArm 233MHz CPU's) for executing jobs during the spare time they pass on a battery loader?

The aims of JXTA project is to provide support for Peer-to-Peer communications over Firewire and IR connections, while the Jini project (SUN), provides a framework for integration of coffee-machines, refrigerators and other household machines into one unique network. We can reasonably presume that whatever CPU resides in our futuristic refrigerator, it will most of its time be unused.

The Peer-to-Peer protocols provide an intuitive way of auto-recognition between those potential CPU resources. What those systems lack is a social characteristics essential in a good collaboration between this new kind of entities. The MAS paradigms provide a huge amount of rules and social behaviors among which we can choose in order to make act and interact our Peers.

One should understand that the social behaviors spotted in animal or human societies are an invaluable source of informations which should not be neglected when we talk about resource or job sharing. Although it is true that we cannot understand completely those behaviors, we can model them and use as the starting point in designing communication protocols involved in task exchange.

P-ICE is only a starting point of our project. It provides an Peer-to-Peer architecture without going deeply in interactional aspects of resource sharing. In future work we should experiment different social behaviors of Peers in order to achieve even higher performance. Is the society in which entities ask for job better than the society in which entities impose a job? *(pull* or *push* model?). Is the mix of two societies better than the two separately considered societies? MAS approaches base on studies in different scientific fields ranging from ethology to sociology trough a wide range of other disciplines.

Unfortunately the Multi-Agent Systems are used in major part for simulation of biological phenomena and not for providing a consistent framework for world wide computer interactions. The unification of resources, services and demands is the key of future development in computer science. Lot of work in this direction has been done the past few years, and even more is to be done. The further step is the integration of all computational and communicational resources in one world-wide cluster. Considering that a whole is greater than a sum of all it's parts, the future seems promising.

10. Conclusion

Our distributed computing environment, P-ICE, has fulfilled the requirements. The system is efficient, scalable and fits very well for digital image processing applications. P-ICE is stable and able to resist the hardware/OS failures.

Even if P-ICE is still a prototype, and not a commercially exploitable application, the results of the measured performance provide a proof of concept for the peer-to-peer distributed computing. The efficiency of almost 80% shows that it is possible to obtain an effective distributed system in highly heterogeneous environments. Using Peer-to-Peer approach, we have obtained a system which scale very well and which adapts rapidly to changing environment.

We have not tested P-ICE on a huge number of computers, but the framework is flexible enough to allow experimentation and development of new algorithms. The informations about the topology of the network, obtained by network multicasts, add and adaptive behavior to the whole system. This is valuable when the processor load vary constantly.

The advantages of an efficient distributed computing environment are clear. If we consider that the cost of developing and manufacturing an integrated distributed system grows almost exponentially with the amount of processors, the alternative solutions as ours, which keep this dependency linear, must be envisaged. While it is true that the performance of the second is lesser, their financial advantage should not be neglected.

Moore's Law states that the performance of computers doubles every year. Buying 10 computers now, in two years and considering an efficiency of 80%, we will have an equivalent of 4 new computers, leading to an investment return of 40%, which is considerable having in mind high amortization rate on computer hardware.

The drawback of such low-cost systems is that they rely on cheap and unstable hardware. This should be not an excuse for the poor performance or scalability issues. Such systems, if constructed carefully, can handle huge tasks and compete with integrated, massively parallel systems.

Part I.

Appendix

A. Known Bugs and *To do* List

A.1. Performance increase issues

A.1.1. Socket latency

When the Peer sends the data trough the actual implementation of the socket, and the receiver socket is not opened (in order to protect from buffer overload or due to an OS crash), the receiver Peer is recontacted several times. After the defined number of tries, the packet is dropped. The loss of packet is not a problem, while the Get and Send Peer take care of successful arrival of data. The real drawback is performance loss due to this retries. In order to suppress those, the Peer should *directly* send the data to an another available Peer when faced to a closed receiver socket.

One should note that the loss of the image data is not acceptable for the Send Peer, which uses the same socket implementation. That is why the Send Peer first writes in his scheduled entries table that the image has been sent. As the Get Peer do not confirm the arrival of that data, Send Peer re-injects it.

A.1.2. Frequent Broadcasts on huge network

As the number of Peers grows, the broadcasts become more frequent. This not only saturates the network bandwidth but also slows the computing by a Peer, because the Peer must handle the broadcasts received.

Although we provide an interface for implementing efficient Broadcast Strategies, our prototype doesn't use it extensively. Those strategies should in future base their broadcast decision on number of Peers. If this number grows, broadcasts should be done less frequently.

The problem of less frequent broadcast resides in fact that in that case the perception of Peers will be less accurate. When the further work on P-ICE is done, we should on one hand optimize the Broadcast Strategies in an above-mentioned manner, and on the other, we should develop more sophisticated Distribution Strategies, based on past states of the system. This should allow to predict the behavior of Peers and compensate the missing information about environment.

A.1.3. Computing Peer [Check Load & Distribute] Task

The [Check Load & Distribute] Task assures that the distribution of data is done uniformly in the Peer network. The Peer, when a specific condition is met, injects some of its job in the environment. In our prototype this condition is satisfied when the number of jobs in the Watermarker input queue exceeds 5. When this happens, we use the "Smallest Input Queue" Distribution Strategy in order to reinject the data in the environment.

This strategy suits very well for the initial (Send Peer) injection, but in the Sub-Optimal Peer number environments (Figure 6.3 on page 28) it generates significant overhead. As lot of Peers have their buffers full, this reinjection occurs very often.

Better strategy would be to choose to reinject only if the candidate Peer has significantly less available work as the demanding one.

A.2. *To do*'s

A.2.1. Get and Send Peer Crashes

There is no lot to do in order to avoid the crash of the Send and the Get Peer. Even if we didn't experience this during our tests, it could occur. What however we can do is assure that the application doesn't crash the whole computer system, which is often the file server too.

If the Send Peer crashes, the Computing Peers will finish the pending jobs and suspend their activity. On the contrary, if the Get Peer crashes, the Send Peer, as it doesn't receive any confirmation of the successfully processed jobs, will continue to fill it's *waiting table* (Section 5.3.5) with job's to be done. Inevitably this will provoke the crash of the Send Peer. In order to avoid that, we should limit the size of the Send Peer's *waiting table.* Scheduler Peer should be aware of this too.

A.2.2. Update of File system descriptors

Scheduler GUI should provide a button which initiates the update of File descriptors from the Send Peer. For now, this update is done only during the initialization phase of the Scheduler Peer.

A.2.3. UDP Packet size

The Multicast Socket uses the UDP Protocol for sending data. Java's multicast socket needs to know the size of the data to be received in order to open the connection. We have fixed this size to an arbitrary number. However, no test is made in order to assure that the packet which is sent does not exceed this size. As the perceptions which are broadcasted can vary in their size due to past-states table growing, we should do some checks before broadcasting.

A.3. Known Bugs

Independently of numerous memory leaks of the Watermarking DLL, the next message appears sometimes:

```
  Not enough memory for random number list!
  Terminating program!
```

After this message the Peer exits. It is possible that this occurs because we initialize java.util.Random object only once.

B. Performance measures

B.1. Serial Performance

Table B.1 gives the measures obtained during the serial performance tests.

B.2. Parallel Performance

Tables B.2 to B.6 show the measures obtained during the parallel performance tests.

Table B.1.: Serial Performance measurements

Test No	Peer1	Peer4	Peer5	Peer6	Peer7
1	21m16s	13m34s	13m30s	14m05s	12m56s
2	21m57s	13m33s	13m23s	13m55s	12m49s
3	22m06s	13m38s	13m24s	13m51s	12m46s
4	21m55s	13m36s	13m22s	13m49s	12m44s
5	21m09s	14m27s	13m26s	13m53s	12m43s
6	20m11s	14m51s	13m25s	14m02s	12m44s
7	20m16s	14m27s	13m27s	13m52s	12m45s
8	20m28s	13m40s	13m27s	13m52s	12m46s
9	20m10s	13m32s	13m20s	13m49s	12m47s
10	20m13s	14m08s	13m21s	13m58s	12m44s
Average	**21m18s**	**13m56s**	**13m24s**	**13m49s**	**12m46s**
Per image Average	**2.283s**	**1.494s**	**1.437s**	**1.481s**	**1.368s**
Overall Average					**15m02s**
Overall per Image Average					**1.612s**

Table B.2.: One Computing Peer measurements

Test No	Peer1	Peer4	Peer5	Peer6	Peer7
1	22m04s	14m19s	13m51s	14m18s	13m13s
2	22m51s	13m16s	13m55s	14m07s	13m37s
3	21m20s	14m12s	13m35s	14m12s	13m17s
4	21m57s	14m12s	13m46s	14m09s	13m18s
5	21m20s	14m40s	13m42s	14m27s	13m29s
6	21m55s	15m59s	13m42s	14m15s	13m13s
7	21m19s	14m45s	13m44s	14m17s	13m30s
8	22m02s	13m48s	13m34s	14m06s	13m14s
9	21m24s	14m15s	13m43s	14m13s	13m10s
10	22m03s	14m16s	13m32s	14m18s	13m14s
Average	**21m49s**	**14m22s**	**13m42s**	**14m14s**	**13m19s**
Per image Average	**2.338s**	**1.539s**	**1.469s**	**1.525s**	**1.427s**

Table B.3.: Two Computing Peers measurements

Test No	Peer1+4	Peer4+5	Peer5+6	Peer6+7
1	9m08s	7m28s	7m34s	7m19s
2	9m08s	7m27s	7m32s	7m17s
3	9m10s	7m27s	7m29s	7m18s
4	9m13s	7m24s	7m29s	7m21s
5	9m18s	7m28s	7m29s	7m17s
6	9m08s	7m21s	7m26s	7m18s
7	9m07s	7m24s	7m26s	7m19s
8	9m06s	7m29s	7m22s	7m19s
9	9m11s	7m29s	7m29s	7m18s
10	9m13s	7m30s	7m30s	7m20s
Average	**9m10s**	**7m26s**	**7m28s**	**7m18s**
Per image Average	**0.9825s**	**0.7976s**	**0.801s**	**0.78321**
Serial Average	**18m05s**	**13m40s**	**13m36s**	**13m17s**
Speedup	**1.9729**	**1.8356**	**1.8214**	**1.8171**
Average Speedup				**1.86175**
Efficiency	**0.986**	**0.9178**	**0.910**	**0.9085**
Average Efficiency				**0.930575**

Table B.4.: Three Computing Peers measurements

Test No	Peer1+4+5	Peer4+5+6	Peer5+6+7
1	6m03s	5m18s	5m20s
2	6m01s	5m20s	5m22s
3	5m54s	5m27s	5m18s
4	5m56s	5m25s	5m14s
5	5m58s	5m14s	5m18s
6	6m02s	5m28s	5m23s
7	6m01s	5m12s	5m17s
8	6m00s	5m14s	5m20s
9	6m01s	5m26s	5m19s
10	5m55s	5m21s	5m13s
Average	**5m59s**	**5m20s**	**5m18s**
Per image Average	**0.641s**	**0.572s**	**0.568s**
Serial Average	**16m12s**	**13m43s**	**13m19s**
Speedup	**2.7067**	**2.5678**	**2.512**
Average Speedup			**2.5955**
Efficiency	**0.9022**	**0.8559**	**0.8375**
Average Efficiency			**0.8652**

Table B.5.: Four Computing Peers measurements

Test No	Peer1+4+5+6	Peer4+5+6+7
1	4m41s	4m06s
2	4m40s	4m08s
3	4m27s	4m06s
4	4m38s	4m09s
5	4m41s	4m12s
6	4m31s	4m11s
7	4m38s	4m08s
8	4m31s	4m14s
9	4m39s	4m18s
10	4m39s	4m11s
Average	**4m36s**	**4m10s**
Per image Average	**0.4937s**	**0.4469s**
Serial Average	**15m36s**	**13m28s**
Speedup	**3.385**	**3.2281**
Average Speedup		**3.30655**
Efficiency	**0.8462**	**0.807**
Average Efficiency		**0.8266**

Table B.6.: Five Computing Peers measurements

Test No	Peer1+4+5+6+7
1	3m47s
2	3m43s
3	3m46s
4	3m54s
5	3m41s
6	3m47s
7	3m46s
8	3m42s
9	3m48s
10	3m49s
Average	**3m46s**
Per image Average	**0.404s**
Serial Average	**15m02s**
Speedup	**3.986**
Efficiency	**0.7971**

Index

Bibliography

[1] Corba homepage. http://industry.ebi.ac.uk/ corba/.

[2] Jxta homepage. http://www.jxta.org.

[3] Message passing interface homepage. http://www-unix.mcs.anl.gov/mpi/.

[4] Parallel virtual machine homepage. http://www.epm.ornl.gov/pvm/pvm home.html.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Proc. of the SJCC*, 31:483–485, 1967.

[6] E. G. Coffman, Jr., M. J. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(1):67–78, March 1971.

[7] S. Craver, N. Memon, B. L. Yeo, and M. M. Yeung. Resolving rightful ownership with invisible watermarking techniques: Limitations, attacks and implications. *IEEE Journal on Selected Areas in Communications*, 16(4):573–586, May 1998.

[8] Frédéric Deguillaume, Sviatoslav Voloshynovskiy, Shelby Pereira, Maribel Madueño, and Thierry Pun. Filigranage d'images digitales. Bull. ASE/SEV, April 2001.

[9] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569–??, September 1965.

[10] Jacques Ferber. *Les Systèmes Multi-Agents*. InterEditions, Paris, 1995.

[11] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors, Vol. I.* Prentice–Hall, Englewood Cliffs, NJ, 1988.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] J. W. Havender. Avoiding deadlock in multiasking systems. *IBM Systems Journal*, 7(2):74–84, 1968.

[14] Alexander Herrigel. Eigentum schützen mit wasserzeichen. *Computerworld*, (6):A14–A16, 8 February 1999. (special issue: Internet).

[15] High performance fortran language specification. *Scientific Programming*, special issue(2), 1993.

[16] Vipin Kumar. *Introduction to Parallel Computing.* Addison-Wesley, 2001.

[17] Ted G. Lewis, Hesham El-Rewini, and with In-Kyu Kim. *Introduction to Parallel Computing.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[18] Sheng Liang. *The Java Native Interface, Programmer's Guide and Specification.* Addison-Wesley, 1999.

[19] Shelby Pereira. *Robust Digital Image Watermarking.* PhD thesis, Computer Vision Group -CUI - University of Geneva, Geneva, Switzerland, 2000.

[20] Fabien Petitcolas. Stirmark. http://www.cl.cam.ac.uk/ fapp2/watermarking/stirmark/.

[21] Guy Robinson. Parallel computing works. *Parallel Computing Works*, 1995. www.npac.syr.edu/copywrite/pcw/.

[22] Sviatoslav Voloshynovskiy, Shelby Pereira, and Thierry Pun. Watermark attacks. In *Erlangen Watermarking Workshop*, Erlangen, Germany, 5–6 October 1999. (invited presentation).