Wide scale distributed processing using connected limited devices

R.M.Vaandrager
Universiteit van Amsterdam
rmvaandr@cs.vu.nl

July 10, 2002

Abstract

This paper will focus on the use of Java 2 Micro Edition (J2ME) compatible Connected Limited Devices (CLDs) such as mobile phones, PDA's, set-top boxes and consoles in a wide scale distributed processing environment. Discussed are (amongst others) requirements, standards and architectures needed to provide a solid platform for CLD distributed processing. A working proof of concept application will illustrate how to implement such a system and highlight the problems that arise. Limitations and bottlenecks will be identified and solutions given if available. Finally we will cover how this field might develop in the future as technology rapidly advances.

Contents

1 Distributed Computing: Introduction

1.1 Definition

> "A computer system in which several interconnected computers share the computing tasks assigned to the system." *IEEE(90)* [5]

1.2 Principle

the main principle of distributed computing is to make more than one machine work on the same problem simultaneously. This has several advantages:

1. greater computing speed as many can do what one cannot.
2. higher reliability as the computation is spread over a large number of geographically widely dispersed systems lowering the risk of system faillure.
3. lower operating cost as maintenance, hardware upgrades, storage and power consumption cost are spread among the many owners of participating units ensuring an easy to maintain system with virtually no obsolescence.

1.3 Applications

These benefits and recent technological and economical developments such as the mass acceptance of personal computers and the internet as a communication medium have resulted in distributed computing being used on wide scale and in many fields of specialisation ranging from math to life sciences to games.

Table 1 gives an incomplete list showing some current active distributed computing applications on the internet.

pic

1.4 Characteristics

A good distributed computing system has a number of characteristics:

Name	Function
Distributed.net	Brute force blockcypher decryption
SETI@Home	Search for extraterrestrial intelligence
Golem@home	Evolutionary robotics
Folding@home	Human-genome protein folding
Muon1	Particle acceleration design
the distributed chess project	Chess
Rutherford Appelton Laboratory	Climate prediction

Table 1: Some active distributed computing applications.

1. Generality: of a system refers to the possibility for the system to adapt to new problems and the portability of the system in general (Platform independent).

2. Accessibility: means that the available computing resources must be easily accessible at the lowest possible cost.

3. Maintainability: a distributed computing must be easy to maintain despite of the number of clients and underlying network complexity.

4. Scalability: The system should easily be able to encompass a large number of nodes and be able to quickly deal with large fluctuations in available resources.

5. Reliability: The system should be stable and perform correctly with a high level of uptime. (Fault-tolerance and Recovery).

6. Security: The system has to be vulnerable against tampering. as a large number op nodes are present involving many users over a relatively insecure networks such as the internet it is of high importance that the system is secure and calculations and communications have a very high level of integrity. Use of encryption, double or triple checking of results against multiple nodes and HTTPS will help to improve overall security. (Unified security model)

1.5 Problem space

Not all problems are suited for distributed computing: dependant on granularity, priority, coupling, sequentially. Problem suitable to be solved by distributed computing need to have a number of characteristics.

1. Dividable: The problem has to be easily dividable in smaller sub problems or tasks.

2. Sequential: These tasks have to be non-sequential and loosely coupled. This means that there is no need to solve earlier tasks in order to solve the following task and that these tasks can be completed on the client system with a minimum or no interaction with other tasks running on different clients.

3. Granularity: The tasks need to have the right granularity suited to the distributed computing system architecture used to solve the problem. Granularity of a system refers complexity and time needed to compute the task into a final result. Fine grained problems are very small and simple tasks where course grained problems require a large amount of computational power to reach a result. In general large scale distributed computing systems require course grained problems.

4. Data-to-compute-to-data ratio: The tasks need to have a reasonable data to compute and compute to data ratio. This means that the size in bytes of the unsolved task sent to the client has to be small in comparison to the computational power needed to solve the task. The compute to data ratio means that the final result of the computation also has to be small in comparison to the computational power needed to solve that particular task.

5. Priority: Finally the process priority is important. If the problem has to be solved in real time it usually is not suited for wide-scale distributed computing due to the course-grained granularity of the sub-tasks. Low process priority (results can be obtained in a matter or days or even years) however are ideally suited for distributed computing.

1.6 Distributed Computing Systems

As the definition given by IEEE is rather general in nature and a lot has happened in the field of distributed computing since it's formulation. We would

like to make a classifications of different distributed computing systems based on the following characteristics: network communication topology, application type and resources involved.

Based on these characteristics there are roughly three different kind of distributed computing systems:

1. A distributed processing system. This systems is characterized by having a centralized management organisation (client server), minimal communication between the client and the server and no communication between clients. (comm top. Application, resources.)

2. A distributed computing system. This system also has a very centralized topology, but since the application is interactive in nature it requires a higher degree of Client-Server communication and also limited Client-client communications. Resources donated are processor cycles and memory.

3. A grid computing system. This is a very decentralized distributed computing system in which every client can also be a server. Applications can range from batch processing to interactive programs. All resources can be shared (CPU, mem, data, instruments)

As the complexity and bandwidth requirements of these system increases there is a bigger demand for services to ensure correct operation and easy maintainability of the system as shown in Figure 1. These services include: Scheduling, Process Migration, Distributed Files system, Name and directory services

1.7 Device and system layers

As the internet becomes part of daily live for many people there is clearly visible trend to connect many kinds of devices to the internet, as the number and heterogeneity of the connected devices growth new ways to use these resources for distributed computing become possible. This creates a large pool of currently untapped processing resources that might be used for distributed processing.

Figure 2 shows four different types/configurations of parallel computing systems and their respective characteristics.

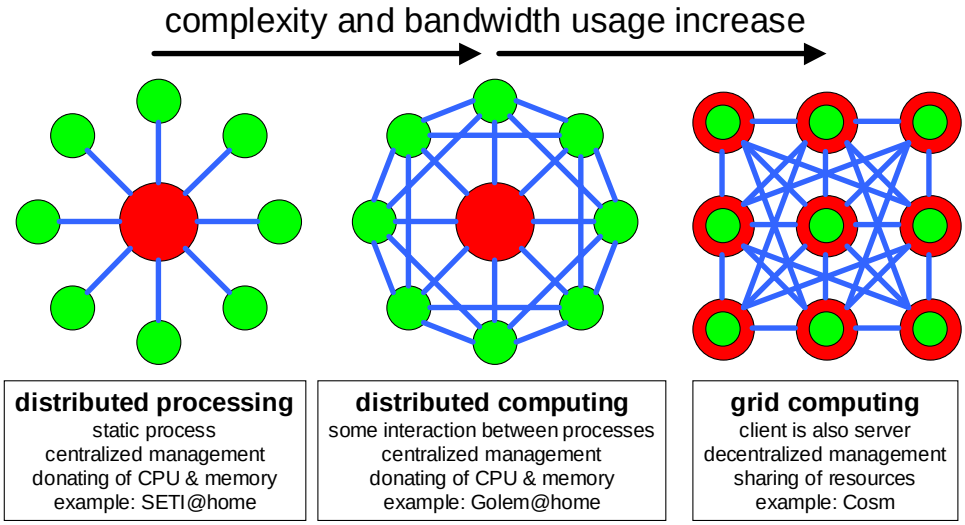

Figure 1: Distributed computing systems.

1. At the core we can find supercomputers. Calculations done on these systems are often distributed in nature, and these systems generally consist of a large number of high-speed processors on a fast internal backplane. However geographically the distribution of tasks is typically limited to one machine. Calculations done on these machines often have real-time priority such as virtual reality modelling in a cave environment and therefore a very fine grained task granularity. The Cost of these systems is very high and will quickly move into obsolescence.

2. As we move to the outer layers the number and heterogeneity of devices participating in the system increases as does the geographical scale in which calculations take place. The second layer consists of high-end workstations clustered together on a fast small scale network, often a LAN. These systems are relatively cheap compared to their supercomputer counterparts and can be easily replaced and upgraded as needed. These systems are well suited for semi-real time applications such as search engines where a result is typically needed in under 5 seconds. Problem granularity is thus fine to medium grained.

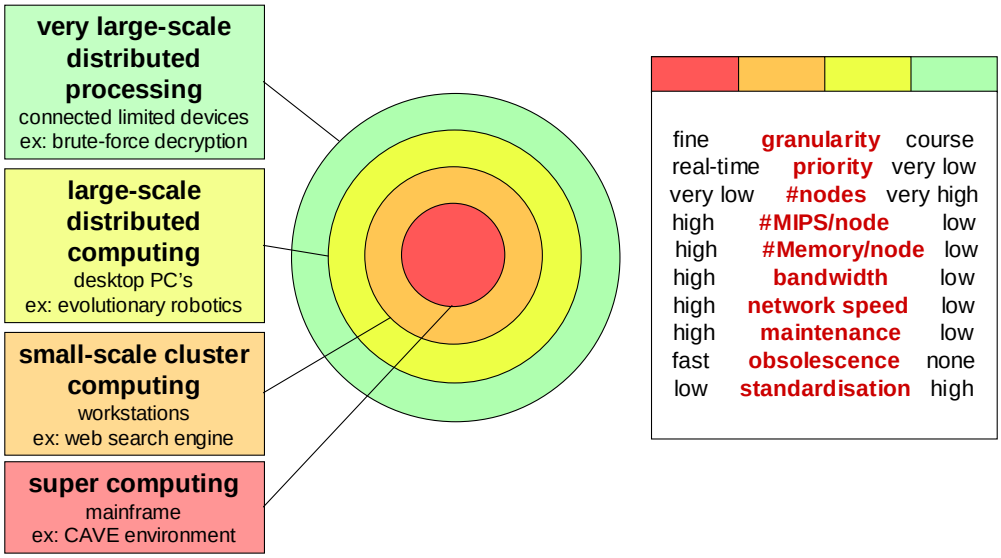

Figure 2: Processing devices and systems pool.

3. The third layer moves distributed computing out to wide area networks (WAN) and the internet. One can think of an international organisation linking their installed desktop computer base distributed over multiple geographically dispersed branches for distributed computation or open distributed computing projects where all connected desktop computers can participate in the computing effort. This system is what most people see as distributed computing due to the huge popularity and media coverage on projects such as SETI@home. Applications running on these systems distribute tasks that are course grained in nature and therefore have a low process priority, task results will typically be expected in a matter of hours or days. Systems participation in these projects are often low to medium-end desktop computers that are frequently (every 2 years or so) replaced by newer and faster models by the respective owner. These systems are often outside of the reach of the distributed system maintainer so there is no cost involved in expanding and keeping the system up to date. One could say these systems will evolve over time and will not or very slowly become obsolete. Security becomes a big issue in these widely distributed systems.

4. The fouth and final layer shows connected low-end devices such as mobile phones, pda's, set-top boxes and game-consoles. These devices are different from traditional devices connected to the internet such workstations and pc's in the sense that they are often limited in both capabilities and number of uses. We therefore call these devices Connected Limited Devices (CLD's). CLD's are many and are usually very frequently (6 month to 1 year) replaced by users with newer and more powerful models. Due to the very homogenous nature of these devices currently no efforts exist to use these devices for the purpose of distributed computing. However since the number of CLD's is rapidly growing and so are their computing resources this is logical a promising area for the expansion of distributed computing systems and thus this paper focuses on distributed computing on devices in the CLD layer in a distributed processing system topology.

2 CLD Distributed Processing

2.1 Device limitations

CLD's typically have a number of limitations when compared to personal computers that will make them less suitable for distributed computing:

1. limited processing power: where a typical pc (500Mhz PIII) will execute 1350 MIPS, a CLD will perform at a fraction of this speed, with phones in the 50 MIPS range and PDA's in the 200 MIPS range.

2. Limited memory available for system resources, applications and permanent storage.

3. Limited i/o: network communication is often wireless, low bandwidth with a relatively high cost per packet and further limited by the availability of one or two protocols such as HTTP over TCP/IP or UDP. Information input on CLD is often limited to the use of a keypad, stylus, remote control or game pad. Visual output can range from external TV-monitors, small built in LCD's to LEDS, or be completely absent.

4. Limited power source such as batteries. Further limitations might include the incapability to do multitasking and a lack of floating point support.

2.2 Standards and middleware

To harness a substantial amount of computing resources, a (wireless) distributed processing application has to interact with a very large number of CLD's. Because these devices tend to be very heterogeneous in nature it becomes clear that there is a great need for standardized middleware to manage the complexity and heterogeneity of this kind of distributed infrastructure. 3 Factors are essential for very wide-scale distributed processing on CLD's. These factors are Computing Platform, Communication Protocol and the Data Description Format. The driving technologies behind the standardization of CLD's are: Java, IP and XML. The Java2 Micro Edition (J2ME) delivers a small and virtual machine (KVM) to run platform independent applications written in sun's java on (CLD), IP offers an efficient and robust protocol to communicate to the outside world. XML offers a self-describing language for and platform independent communication of content.

2.2.1 JAVA

Sun Microsystems has adapted their platform independent programming language Java2 for use on these kind of small devices in the shape of the Java2 Micro Edition (J2ME). Sun's J2ME is a subset of their desktop equivalent; the Java2 Standard Editon (J2SE). however in creating this slimmed down version sun dropped a lot of extra features and left only the bare basic functionality for data structures, data types and network support intact. In it's current implementation the J2ME is not capable of handling floating point numbers thus severely limiting it's use for scientific distributed applications which often rely on high precision calculations involving floating points. However this might be fixed in a future version of the J2ME. CLD's that support J2ME usually are installed with the so called Kilo Virtual Machine (KVM) that can run J2ME compatible bytecode. The KVM is called so because the memory it uses is measured in kilobytes in contrast to the full fledged Java Virtual Machines (JVM) found on desktops and workstations. The J2ME environment is shown in Figure 3.

2.2.2 IP

The Transport Control Protocol / Internet Protocol (TCP/IP) is the de facto communications protocol for use on the internet and therefore the protocol of choice for CLD's. A limitation affection the use of IP is that the current

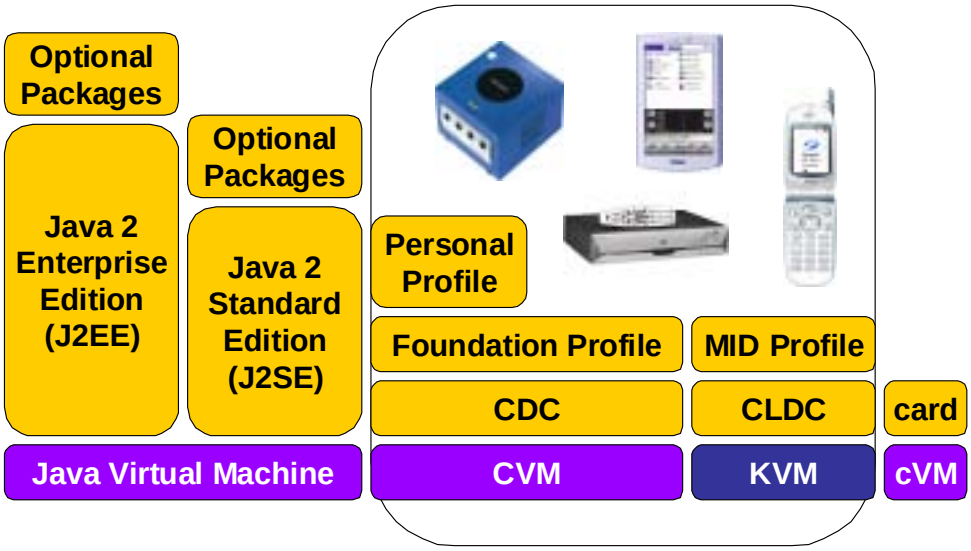

Figure 3: The Java 2 Micro Edition (J2ME) environment.

widely supported version 4 (IPv4) has limited support for addressing the number of devices connected to the internet. Especially CLD type devices will suffer from this fact as these devices are many and it will be hard to give them all a static IP address, The next version of the internet protocol will be Ipv6 that will address this issue and allow for a static IP address to be assigned to all internet capable devices. Although CLD WDP is possible with the use of Ipv4, ipv6 will be the version of choice.

2.2.3 XML

The eXtendable Markup Language provides for a self describing and form independent way of sending information and data between connected entities. xml can be used for anything from describing content, context to remote procedure calls (RPCs).

Together these three technologies create the synergy needed to leverage distributed processing on small connected devices. Figure 2 shows the layers of a distributed system making use of middleware.

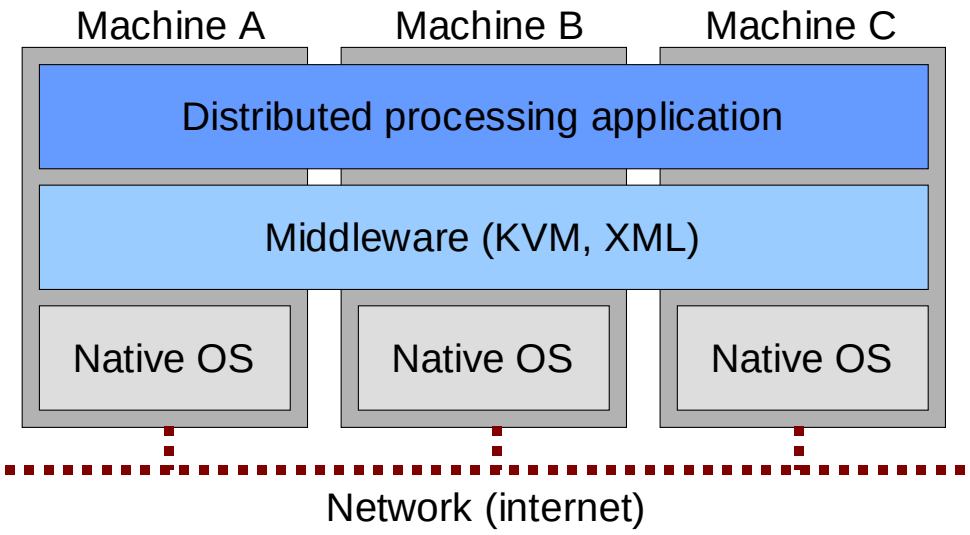

Figure 4: A distributed processing system.

2.3 Psychological factor

It is one thing to create a good infrastructure suited for CLD DP, but another thing to actually make people use it. When it comes to distributed computing efforts outside of controllable boundaries (such as organisations), one has to convince the user to participate in such an effort to be able to uses these resources. First of all the user doesn't want to be hindered in its daily use of the device in question, therefore the application should run at the lowest system priority and transparently to the uses. The user will not run the application if his privacy or system resources might be exposed for uses outside of the will of the user. Therefore trust and security are major issues. Also the project has to be philanthropic or beneficial in nature to appeal to the user.

3 The case for mobile phones in Japan

We will take mobile phones in Japan as an example of a possible implementation of a wireless distributed processing system on CLD's

3.1 Carrier Distribution

In japan there are currently 4 major mobile telecom carriers, these are the formerly state owned NTT DoCoMo, Vodaphone J-phone, and the KDDI-group owned AU and TU-KA as shown in Figure 5.

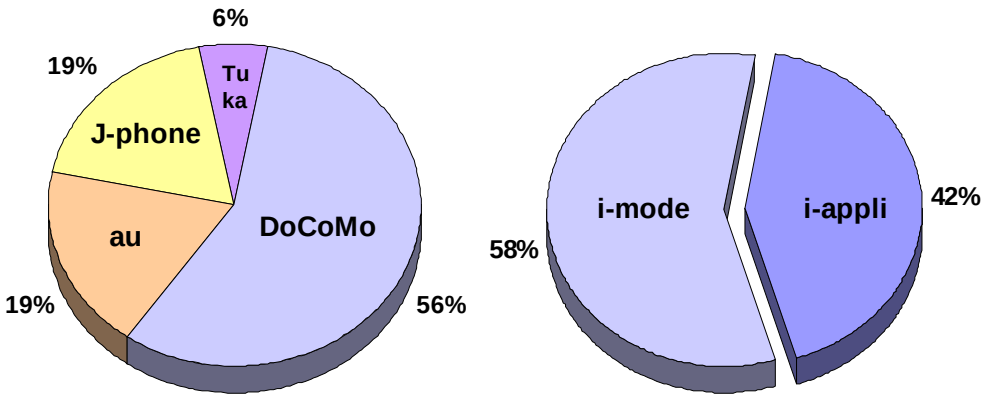

Figure 5: Carrier distribution and percentage of Java capable phones in the DoCoMo network.

All carriers with the exception of TU-KA now offer mobile phones capable of running JAVA programs with the ability to communicate to the internet using HTTP(S). DoCoMo has been one of the earliest adopters of J2ME for use on mobile phones in the world and began selling java capable phones early 2001. Its version of Java bears the name "i-appli" and is in fact a slightly modified (and thus not standard compliant!) version of sun's J2ME. Currently most phones sold by DoCoMo are capable of running I-appli. Because there is a large user base of java phones and readily available specifications and SDK this system will serve as our example case in this paper. Also mobile phones are the most minimal CLDs that are able to run J2ME applications and therefore present the hardest challenge to implement a simple distributed processing system. If it will run on a mobile phone it is relatively safe to assume it will run on other J2ME compatible CLDs that have more computing and memory resources at their disposal.

3.2 Processing power and potential

To make a rough estimation of the total available processing power on internet connected mobile phones in japan compared to internet connected computers see fig x (table?) - population, phones, internet pc - keitai/pc ratio, keitai/pc MIPS ratio, keitai/PC combined processing ratio - theoretical combined processing power mobile internet connected phones outnumber internet connected computers by roughly seven to one. And even though a regular computer is about 27 times faster than a mobile phone it shows that the mobile phone market is currently able to provide a quarter of the total processing power offered by pc's. If one keeps in mind that mobile phones are just one of the CLDs (others are PDA's, set-top boxes, consoles etc.), and the most limited of devices at that. It becomes clear that these kind of devices should not be underestimated and will be able to offer an comparable theoretical processing power to connected personal computers.

3.3 Environment

Let's take a look at what kind of environment mobile phones in Japan operate in

3.3.1 Networks

network speed ranging from 9.6Kb/s in 2.5G networks to 384Kb/s in 3G networks. Cost per packet send over these networks also depends on the network used and the type of information sent (text, multimedia) but the average cost on 2.5G networks is around 0.25 Yen per 128byte packet Phones are able to connect to the internet using the Hyper-Text Transport Protocol (HTTP) over TCP/IP or UDP in a packet switched wireless network. a secure layer is also available (HTTPS). Requests to a remote server will be made by using the Multipurpose Internet Mail Extensions (MIME). There are three different commands possible namely HEAD, GET and POST depending on the type of request made.

3.3.2 Mobile phones

A typical mobile phone will feature a KVM see fig x. to run java applications on. It will offer space for about ten or so applications, each with a

maximum size of 10 KB, and also offer 10KB of permanent memory called the scratchpad for these applications to utilize.

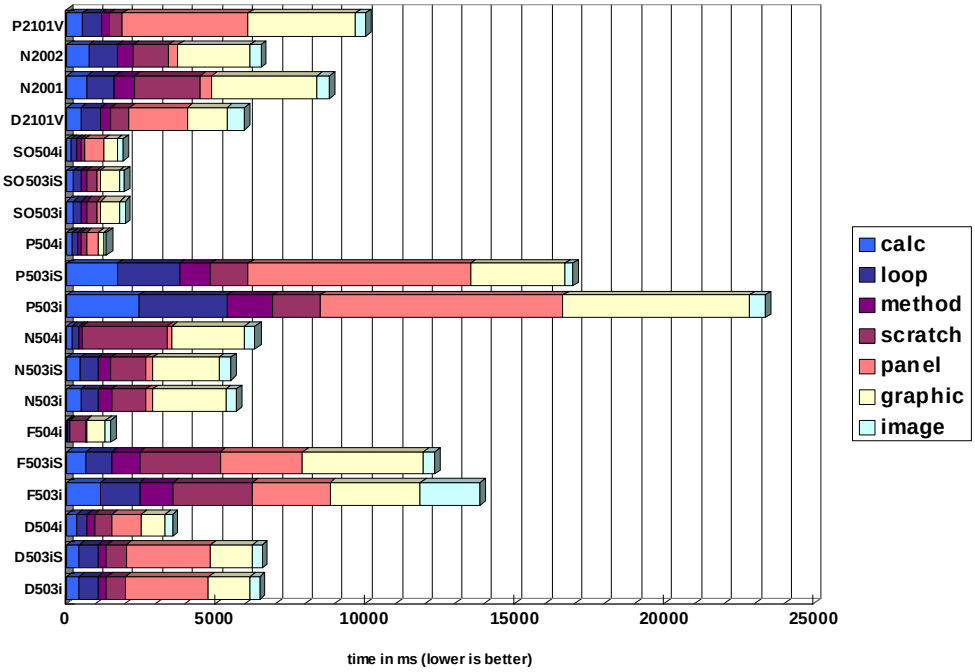

Figure 6: KVM benchmark of DoCoMo models.

Figure 6 shows all I-appli compatible models DoCoMo offers as of July 2002. Represented are 3 generations of models: the 503i series, the 503iS series and the most recent 504i series. Also included are the FOMA (3G) Models. As one can see there are great performance differences between models and generations of models. This difference is sometimes around a factor 20. There are 2 reasons for these big differences. The first is the fact that manufacturers are able to optimize their implementation after successive generations, and the second reason is the move from software based java to software accellerated java using for example the JBlend environment.

Even phones within the DoCoMo network have many different characteristics. Built in LCD range from 120x160px in 8 bit colour to 224x189 in 16 bit colour and processors differ from model to model depending on the manufacturing company.

4 Application

4.1 Purpose

The purpose of the application is to give a proof of concept demonstrating it is currently possible to write a minimal yet working example of a distributed processing application and to analyse the differences in implementation, performance and use compared to a typical distributed computing approach.

4.2 Development

4.2.1 Hardware and software

The application was written for the NEC N503i mobile phone. This is a 2nd generation I-appli phone and one of the most popular models in Japan. To get a rough idea on what can be done on such a system, we will list some specifications of this model: 1) Maximum I-appli size: 10Kb, 2) Permanent storage per application, known as the Scratchpad: 10Kb. Applications running on this phone's KVM can use around 100Kb of working memory. All in all it's just a fraction of the amount of memory that would be available on mainstream desktop pc's.

There are a number of tools available for programming I-appli on this device. Docomo provides development tools such as the DoaJ and API documentation on it's website. A company called Zentek provides a good free N503i emulator for verification and testing.

The proof of concept application in this paper has been written with the DoCoMo DoJa J2ME WirelessToolkit based on the Mobile Information Device Profile (MIDP).

4.2.2 Development path

The Development path for creating I-appli's involves a number of steps:

1. source code compilation to bytecode
2. byte code optimisation and pre-verification
3. archiving of all bytecode and other files such as images and sounds into a Java ARchive (JAR)

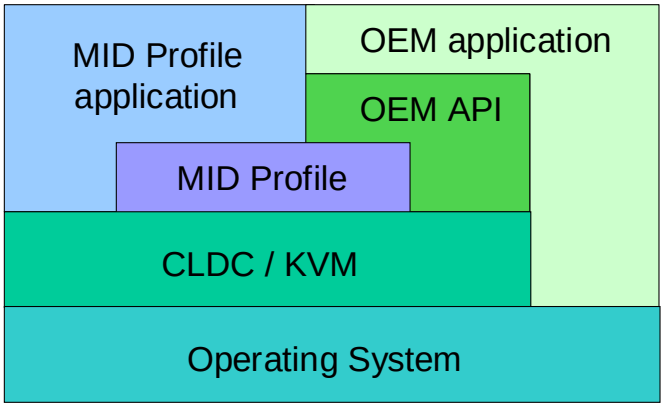

Figure 7: The J2ME framework on a typical CLD.

4. packaging of the JAR by adding a meta description file (JAM)

5. encapsulation of the JAR and JAM files into a HTML or cHTML file.

A user of a compatible mobile phone can now point his I-mode browser to the URL containing these files and download the application to internal memory for use.

4.3 Function

The main function of the application is distributed Mandelbrot-set rendering. The Mandelbrot-set discovered in 1980 by Benoit Mandelbrot is the most well know fractal. A fractal is beautiful geometric shape with a recurring pattern, some of these patterns are often found in nature. Examples include: coastlines, rivers, plant distributions, architecture, wind gusts, music, and the cardiovascular system.

4.3.1 Mandelbrot

The Mandelbrot-set is generated by iterating the formula:

$$Z_{n+1} = (Z_n)^2 + c \tag{1}$$

It represents a connected set of points in the complex plane. Pick a point Z_0 in the complex plane. Calculate: $Z_1 = Z_0^2 + c, Z_2 = Z_1^2 + c, Z_3 = Z_2^2 + c...$

If the sequence $Z_0, Z_1, Z_2, Z_3, ...$ remains within a distance of 2 of the origin forever, then the point Z_0 is said to be in the Mandelbrot set. If the sequence diverges from the origin, then the point is not in the set.

4.4 Architecture and design

The phone can communicate only with the server it was downloaded from using MIME extensions via HTTP(S). there are four different post types: Image/gif, Text/plain, Application/octet-stream and Application/x-www-form-urlencoded.

The application consists of a few major components: the communication and the calculation. The main process loop is shown in Figure 8

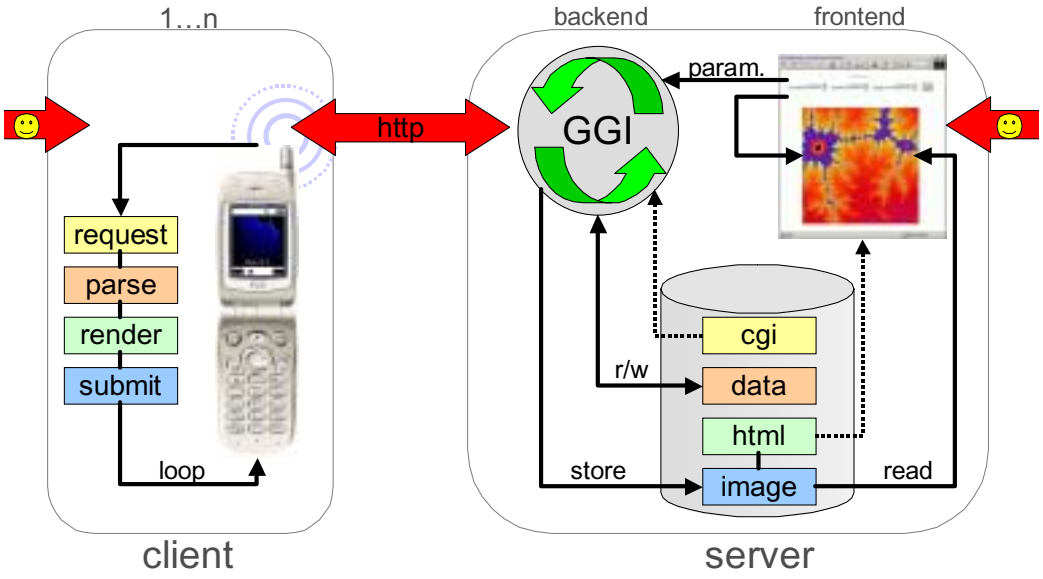

Figure 8: Application design overview.

The application is designed with the Keep It Simple Stupid (KISS) principles in mind. Only the most essential components for a functioning system have been implemented. The main design as pictured in fig x consists of a client and a server side. The Server will distributed tasks known as blocks to the clients for processing. A block is a set of comma-separated parameters that can be parsed by the client.

The client side is written in java and it's main function is the actual computation, it's main process loop is shown below:

1. Request: Block Request by sending a CGI GET request by HTTP to the Server.cgi script on the server. An example request will look like this:

   ```
   GET /cgi-bin/mandel.cgi HTTP/1.0
   ```

2. Parse: Parse the received block, a bock is a plaintext file containing comma separated values in the format: (ID, x, y, z), ID represents the number of the task and x,y,z the coordinates to be calculated in the Mandelbrot-set. An example block will look like this:

   ```
   HTTP/1.1 OK 200
   Content-Length: 1000
   Content-Type: text/plain

   3,435,200,45
   ```

3. Proces: The parameters are used to process the block to the screen in 100x100pixels and 7bit colour, and to a Portable Grayscale Map (PGM)

4. Submit: The resulting PGM is send to the server as a CGI Post in the following format:

   ```
   POST /cgi-bin/upload.cgi HTTP/1.0
   Content-Type: Application/octet-stream

   100 100 34 23 23 95 124....
   ```

5. Repeat.

The server side is divided into a backend and a frontend. The backend contains a number of Common Gateway Interface (CGI) scripts written in Perl that deal with communication from the client and server frontend. The frontend is written in a combination of HyperText Markup Language (HTML) and CGI-forms, and is responsible for displaying the calculation results and controlling the computational process. The Server backend process looks like this: 1)Catch and set parameters 2)Catch request and post first block in queue 3)Catch results, convert to gif, and store.

4.5 Entities and relations

Figure 9 shows all entities and their relations in the application framework.

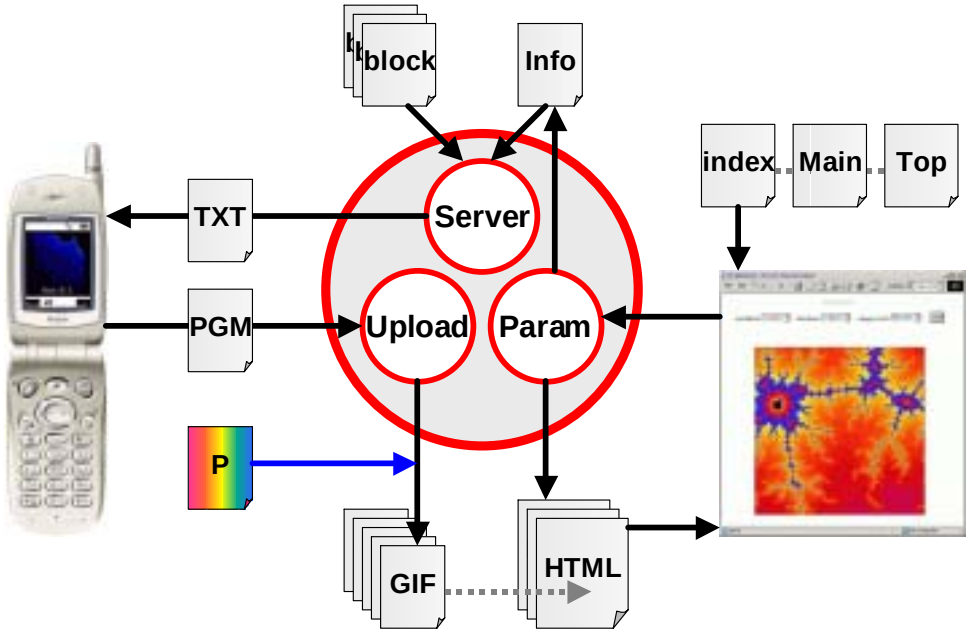

Figure 9: Entities and relations in the application framework.

There are a number of different types of entities:

- JAVA bytecode: Netmandel.java client application
- CGI scripts: Server.cgi Upload.cgi Parameter.cgi
- Plaintext files: information file containing the current block-set and block-counter, multiple blockfiles containing blocks for diffent sets of sizes and a pallete file for substituting the greyscale PGM file with a colour palette.
- HTML files: Frontend consists of 2 frames. one for setting processing parameters and one for displaying the results. the parameter frame influences the results frame.
- Image files: Images send from the client in the PGM format and the converted images in GIF format

4.6 Performance and analysis

4.6.1 Performance and cost issues

If we look at how the finished application behaves we can observe a number of things. First of all we can see that distributed computing is possible, however at a great cost if one considers that the result being send back is an uncompressed graphic file consisting of 100 by 100 pixels in an 8bit colourspace. The filesize of the resulting PGM is therefore around 10Kilobytes large. The cost of sending one 128byte packet out of which roughly 10percent will be headerinformation in the 2.5G docomo network is about 0.25 Yen. A simple calculation shows that the total cost to send one result back from a client to the server is around 85Yen. this is unacceptable for philantropic distributed processing.

Another factor is the time it takes to send this result back. again this depends on the network but to make a pessimistic estimate we will use a 9.6Kb/s network. in this network a 10KB File will consequently take about 8.5 seconds. During this time the client is not able to run computations. One can argue that this might not be a big problem if the task has the right granularity. however on some clients we tested the mandelbrot renders in under a second.

This leads to two fundamental problems with our test application: 1)Task granularity is not course enough and 2) the compute to data ratio is not optimal. Ofcourse other possible applications such as calculating the decimals of pi, finding primes and bruteforce decryption do not have these problems and are better suited for wireless distributed processing.

4.6.2 System scaleability

Although we only simulated the system with a limited number of clients, we expect the system to scale well as communication between the clients and server is limited. Of course our application has a flawed granularity and compute-to-data ratio limiting scaleability but in a typical system a client can be expected to send a result back every 10 hours or so. results might vary from 1KB to 10KB with an average of 5KB. requests are also made every 10 hours and will also be around 1KB in size. Using these figures a host on a T3 internet connection[1] will in theory under optimal conditions be able to serve

[1] T3 speed is 44.736 Mbps (Megabits per second).

34357248 clients, if we substract around 30percent of this capacity to allow for peaks and other limiting factors we can realistically expect this system to serve 24 million active clients.

4.6.3 Lack of floating point support

To accurately generate the mandelbrot-set there is a need for floating point support. because the J2ME MIDP is a stripped down version of java it does not natively support this feature, a possible future version might fix this but at the moment this is a serious problem as many distributed processing applications require high precision calculations[2].

4.6.4 No multitasking in the OS

A distributed processing application should always run in the background without hindering the original purpose and function of the device. (in our case making phone calls and accessing the internet) therefore by the embedded operating system should have support for multitasking allowing multiple programs to run at the same time. this is currently not the case.

4.6.5 Omissions

Due to the limited time and memory avaliable both security and xml standard compliant communication were not implemented. However the implementation of both should be easily possible in a CLD Distributed Processing system.

Secure communications can be achieved by utilizing HTTPS wich is available for use on i-appli compatible phones in the DoCoMo network such as the ones used for the application. figure 10 shows the use of HTTPS in a wireless environment. The information and calculations on the phone can be made secure by using a cypher, this will however affect the already limited processing resources.

XML can provide a self describing method of communication information, and even RPC. XML functionality can be implemented by using one of the many available XML parsers for JM2E compatible devices. examples of parsers are shown in table 2

[2]Currently there is 3rd party support for floats in the J2ME from Jscience.

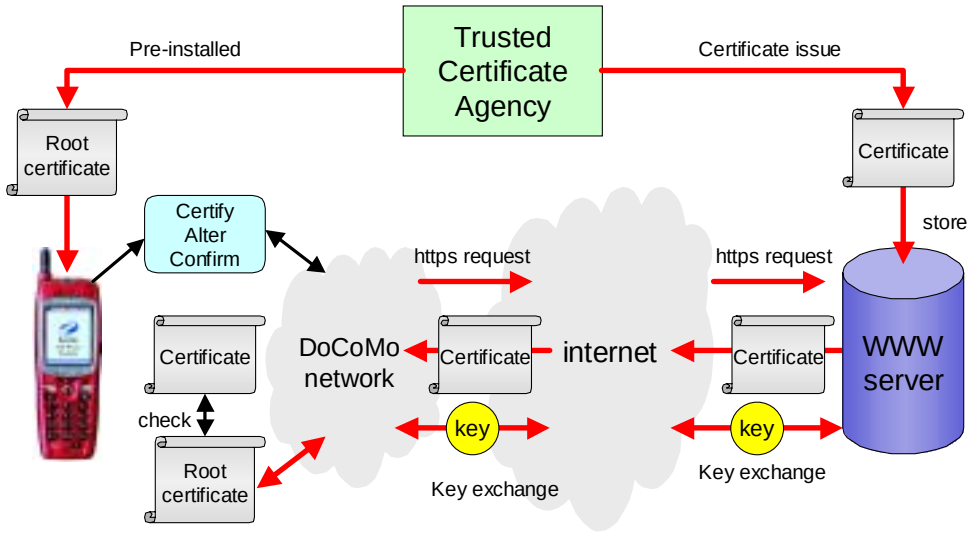

Figure 10: HTTPS based communcation in the DoCoMo network.

to give an example on how an implementation using XML might look we created a fictional Document Type Definiton (DTD) called FraXML for sending over mandelbrot-blocks in the distributed application:

```
<!ELEMENT fraxml(mandelbrot, meta)>
<!ELEMENT mandelbrot (size, range, iterations, period, colors)>
<!ELEMENT size (x, y)>
<!ELEMENT x (#PCDATA)>
<!ELEMENT y (#PCDATA)>
```

Name	Size	MIDP	Type	Sax
kXML	34KB	yes	pull	no
MinML	13KB	no	push	1.0
NanoXML	10KB	patch	model	1.0
TinyXML	13KB	no	model	no
wbxml	19KB	no	push	1.0

Table 2: Small XML parsers.

```
<!ELEMENT range (x, y, z)>
<!ELEMENT x (#PCDATA)>
<!ELEMENT y (#PCDATA)>
<!ELEMENT z (#PCDATA)>
<!ELEMENT iterations (#PCDATA)>
<!ELEMENT period (#PCDATA)>
<!ELEMENT colors (#PCDATA)>
<!ELEMENT meta (model, ID, request, submit, bitmap, timeout)>
<!ELEMENT model (#PCDATA)>
<!ELEMENT ID (#PCDATA)>
<!ELEMENT request (#PCDATA)>
<!ELEMENT submit (#PCDATA)>
<!ELEMENT bitmap (#PCDATA)>
<!ELEMENT timeout (#PCDATA)>
```

a block sent by the server to the client using this FraxML DTD model would look something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<frxml:FXRML>
  <mandelbrot>
    <size>
      <x>120</x>
      <y>120</y>
    </size>
    <range>
      <x>0</x>
      <y>0</y>
      <z>0</z>
    </range>
    <iterations>256</iterations>
    <period>1000</period>
    <colors>256</colors>
  </mandelbrot>
  <meta>
     <timeout>2</timeout>
  </meta>
</frxml:FRXML>
```

Other omissions are a lack of load-balancing, downloading multiple blocks for processing, allowing storage of partial results when interuppted, client-side data encryption, compression of results and a number of other non essential but relatively important features.

5 The future of CLD distributed processing

5.1 Future Requirements

So far we have identified a number of additional requirements needed in the future to make distributed processing a reality:

- Multitasking in the OS: Distributed processing happens in the background with the lowest system priority.
- Flat-fee network access: Cost has to be minimal.
- Floating point support: Many applications rely on high precision calculations.
- Low demand on the Power-source: nobody will use the application if it drains their power source.
- IPv6: Allows all CLDs to have a unique address for identification on the internet and distriubted networks.

5.2 Facts and expectations

- Moore's Law states that for a given cost, the amount of computing power available for that cost doubles approximately every eighteen months.
- J2ME will be leveraged by 70 percent of smart phones and PDAs by 2004[3]
- There is a trend to move Java support in CLDs into hardware resulting in roughly 10x higher performance

[3]Gartner group, nov 2000.

These facts and expectations show that the number as well as the processing power of CLDs will dramatically increase in the future, making CLD based distributed processing a viable solution.

References

[1] J. Knudsen, *Wireless Java: Developing with Java 2, Micro Edition*, 2001.

[2] ASCII Press, *i-mode Java programming: network applications*, 2001.

[3] ASCII Press, *i-mode Java programming: stand-alone applications*, 2001.

[4] R. Riggs, *Programming Wireless Devices with the Java 2 Platform, Micro Edtion*, 2001.

[5] B.B. Mandelbrot, *The fractal geometry of nature*, 1983.

[6] G. Neven, *Implementing Secure Distributed Computing with Mobile Agents*, 2000.

[7] E.T. Rodrigues, *Reliable Computing over Mobile Networks*, 1995.

[8] George Y. Liu, *A Mobile-Floating Agent Scheme for Wireless Distributed Computing*, 1995.

[9] Encyclopedia of Computer Science, *Mobile Computing: Challenges and Potential*, 1996.

[10] I. Foster, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations.*, 2001.

[11] D. Flanagan, *Java in a nutshell*, 1999.

[12] R. Prakash, *Distributed Dynamic Channel Allocation for Mobile Computing: Lessons from Load Sharing in Distributed Systems*, 1995.

[13] Teck-How Chia, *Strategically Mobile Agents*, 1996.

[14] Iso/Iec, *Open Distributed Processing- Reference Model Part 2: Foundations International Standard 10746-2 Itu-T Recommendation X.902*, 1995.

[15] Institute of Electrical and Electronics Engineers, *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, 1990.

[16] L. Erlanger, *Distributed Computing: An Introduction*, 2002.

[17] G. Hewgill, *COSM: Philosophy and Economies of Collective Computing*, 2000.

6 Appendix A Source code

netmandel.java source code:

```
import java.io.*;
import javax.microedition.io.*;
import com.nttdocomo.ui.*;
import com.nttdocomo.util.*;
import com.nttdocomo.io.*;
import com.nttdocomo.net.*;


public class NetMandel extends IApplication {
NetMandelCanvas canvas;

public void start() {
canvas = new NetMandelCanvas(this);
Display.setCurrent(canvas);
canvas.start();
}

static void main(String argv[]) {
(new NetMandel()).start();
}
};

class NetMandelCanvas extends Canvas implements Runnable {
NetMandel caller; // caller

final static int shift = 29;

int id;
int xvalue, yvalue, zvalue;
int table[][];

int curx;
int cury;
// int pwidth = 1;
// int pheight = 1;
int width = 100;
int height = 100;
int max_pixels  = width*height;
int period = 1000;
int max = 127; // max what? iterations?
int max_colors  = 127;
int colors[]  = new int[max_colors];
byte bitmap[]  = new byte[max_pixels];
// String header  = ("id="+id+"&cmd=P5 "+width+" "+height+" "+max_colors+" ");
String recvUrl = IApplication.getCurrentApp().getArgs()[0];
String sendUrl = IApplication.getCurrentApp().getArgs()[1];

Thread curthread = new Thread(this);

NetMandelCanvas(NetMandel caller) {
this.caller = caller;
int red  = 0;
int green  = 0;
```

```
int blue  = 0;

for (int i=0; i<max_colors; i++) { // create the pallete
if (i>=0 && i<=4)  blue+=30;
else if (i>4 && i<=24)  blue+=5;
else if (i>24 && i<=49)  green+=10;
else if (i>49 && i<=74)  blue-=10;
else if (i>74 && i<=99)  red+=10;
else if (i>99 && i<=124)  green-=10;
else if (i>124 && i<=149)  blue+=10;
else if (i>149 && i<=174)  red-=10;
else if (i>174 && i<=199)  green+=10;
else if (i>199 && i<=224)  blue-=10;
else if (i>224 && i<=255)  green-=5;
colors[i] = Graphics.getColorOfRGB(red,green,blue);
}
setSoftLabel(SOFT_KEY_2, "quit");
setSoftLabel(SOFT_KEY_1, "flush");

// width = Math.min(getWidth(), 127);   // previously used for "fit-to-screen" rendering of the image
// height = Math.min(getHeight(), 127);

table = new int[height][width];
}

int count(int x, int y, int max) { //fast int mandelbrot calculation)
int n;
long x0 = x, y0 = y;
long xx = x0, yy = y0;

for (n = 0;
 n < max && (xx * xx + yy * yy) < ((long)4 << (shift * 2));
 n++) {
  long newx;
  newx = ((xx * xx - yy * yy) >> shift) + x0;
  yy = ((xx * yy) >> (shift - 1)) + y0;
  xx = newx;
}
return n;
}

public void proceedDraw() {

Graphics g = getGraphics();
// int totalcount = 0;
// final int limitcount = 5000;
int xoffset =10; // xoffset for image display (top right hand corner)
int yoffset = 5;
int counter = 0;

g.setColor(Graphics.getColorOfName(Graphics.WHITE)); // set font color
g.drawString("Block ID: " + id, 30,120);

while (cury < height) {
int y = yvalue - cury * zvalue;
while (curx < width) {

// if (totalcount >= limitcount) {
// Thread me = Thread.currentThread();
// me.yield();
// if (curthread != me) {
// return;
// }
// }

int x = xvalue + curx * zvalue;
int n = count(x, y, max);

table[cury][curx] = n;
// totalcount += n;
int color = colors[0];
int colorindex;
int black = 60;

if (n < max) {
colorindex = ((n) % colors.length);  // !hack +60 correction for file transfer!
color = colors[colorindex]; // !hack -60 correction for screen rendering!
```

```
n=n+60;
bitmap[counter]=(byte)n;
// bitmap[counter]=(byte)colorindex;
// bitmap[counter]=(byte)color;
counter++;
// SaveInt(colorindex,totalcount);
}else{
bitmap[counter]=(byte)black;
counter++;
}
g.setColor(n); //use built-in pallete
g.setColor(color);
g.fillRect(curx+xoffset, cury+yoffset, 1, 1);
curx++;
}
curx=0;
cury++;
}
//sendImage(); // careful!
}

/*
    public void SaveInt(int n,int pos){

DataOutputStream out = null;
try{
out = Connector.openDataOutputStream("scratchpad:///0;pos="+pos);
out.write(n);
out.close();
}catch(Exception e){e.printStackTrace();}
}
*/

public void run() {
proceedDraw();
}

public void start() {
getRange();
// sendImage();
curthread.start();
}

public void getRange() { // get range from server, parse input and set range variables

//String receiveUrl = IApplication.getCurrentApp().getArgs()[0];

try {
int             respC;
HttpConnection  httpC;
String          respM;
long            contL;

httpC = (HttpConnection)Connector.open(recvUrl, Connector.READ, true);
httpC.setRequestMethod(HttpConnection.GET);
httpC.connect();

respC = httpC.getResponseCode();
respM = httpC.getResponseMessage();
contL = httpC.getLength();

if(200 <= respC && respC < 300) {

char recvData[] = new char[(int)contL];
InputStreamReader inpSr = new InputStreamReader(httpC.openInputStream());
inpSr.read(recvData, 0, (int)contL);
inpSr.close();
String tmpStr = new String(recvData);

int fromIndex=0;
int endIndex;

// read ID
endIndex = tmpStr.indexOf(",",fromIndex);
id = Integer.parseInt(tmpStr.substring(fromIndex,endIndex));
fromIndex = endIndex+1;
```

```
// read X value
endIndex = tmpStr.indexOf(",",fromIndex);
xvalue = Integer.parseInt(tmpStr.substring(fromIndex,endIndex));
fromIndex = endIndex+1;

// read Y value
endIndex = tmpStr.indexOf(",",fromIndex);
yvalue = Integer.parseInt(tmpStr.substring(fromIndex,endIndex));
fromIndex = endIndex+1;

// read Z value
endIndex = tmpStr.indexOf("\n",fromIndex);
zvalue = Integer.parseInt(tmpStr.substring(fromIndex,endIndex));
fromIndex = endIndex+1;

} else {
//do nothing
}
httpC.close();
} catch (Exception e) {
// do nothing
}
}

public void sendImage() {

// byte bitmap[] = new byte[max_pixels];
//
// int color_counter  = 60;
// int index;
//
// for (index=0;index<max_pixels;index++){
// bitmap[index]=(byte)color_counter;
// if(color_counter<159) color_counter+=1;
// else color_counter = 60;
// }

try {
HttpConnection httpC;
int            respC;
String         respM;
long           contL;
String sendData = "id="+id+"&cmd=P5 "+height+" "+width+" "+255+" ";


httpC = (HttpConnection)Connector.open(sendUrl, Connector.WRITE, true);
httpC.setRequestMethod(HttpConnection.POST);
httpC.setRequestProperty("Content-Type", "application/octet-stream");

OutputStream outSt;
outSt = httpC.openOutputStream();
outSt.write(sendData.getBytes());
outSt.write(bitmap);
outSt.close();


httpC.connect();
respC = httpC.getResponseCode();
respM = httpC.getResponseMessage();
contL = httpC.getLength();
httpC.close();
} catch (Exception e) {
// do nothing
}
}


/*
public void sendImage(){

try {
OutputStream outSt;
HttpConnection  httpC;
int             respC;

String sendUrl = IApplication.getCurrentApp().getArgs()[1];
```

```
httpC = (HttpConnection)Connector.open(sendUrl, Connector.WRITE, true); //set mode and url

// send header
httpC.setRequestMethod(HttpConnection.POST);
httpC.setRequestProperty("Content-Type", "text/plain"); // plain text?

outSt = httpC.openOutputStream();
outSt.write(header.getBytes());
outSt.write(bitmap);
outSt.close();

httpC.connect(); // open http communication connection to server
httpC.close();

}catch(IOException e){e.printStackTrace();}
}
*/

public void paint(Graphics g) { // create a black background
g.lock();
g.setColor(Graphics.getColorOfName(Graphics.BLACK));
g.fillRect(0,0,getWidth(),125);
g.unlock(true);
}

public void processEvent(int type, int key) {
if (type != Display.KEY_RELEASED_EVENT) return;

switch (key) {
case Display.KEY_SOFT1: sendImage(); caller.start(); break;
// case Display.KEY_SOFT1: caller.start(); break; // is this a good way, or will it create overflow?
case Display.KEY_SOFT2: caller.terminate(); break;
}
}
}
```

Server.cgi:

```
#!/usr/local/bin/perl

$counter_file="counter.txt";

# open the counter file and create the counter and maximum blocks variables
open(COUNT, $counter_file) || die ("counter file error");
@counter=<COUNT>;
close(COUNT);

($count,$max_blocks)=split(/,/,$counter[0]);

$data_file="$max_blocks.block";

# open the block file and create a block array
open(DAT, $data_file) || die("block file error");
@block_data=<DAT>;
close(DAT);

# ($id,$xc,$yc,$zc)=split(/,/,$block_data[$current_block]);

# calculate and print the current block to be distrubuted according to the counter
$current_block = ($count%$max_blocks)-1;

print "Content-type: text/plain;\n\n";

print "$block_data[$current_block]\n";

# increment the counter by one and store in counter file
$count+=1;
open(UPDATE, ">$counter_file") || die ("counter file update error");
print UPDATE "$count,$max_blocks";
close(UPDATE);
```

preset.cgi:

```
#!/usr/local/bin/perl
```

```
require "cgi-lib.pl";

&ReadParse(*input);

$counter_file="counter.txt";
$preset= $input{'preset'};
$output_file="$preset.html";

open(UPDATE, ">$counter_file") || die ("counter file update error");
print UPDATE "1,$preset";
close(UPDATE);

print "Content-type: text/html\n\n";
print <<ENDHTML;

<HTML>
<SCRIPT LANGUAGE="JavaScript1.2">
<!-- Begin
document.location.href = '$output_file';
// End -->
</script>
</HTML>

ENDHTML
```

sendfile.cgi

```
#!/usr/local/bin/perl -w

use CGI;

$cgi = CGI::new();
$id = $cgi->param('id');
$cmd = $cgi->param('cmd');
$upload_dir = "/home/mich/public_html/cgi-bin/uploads";
$res="done";
$size = length($res);

print <<"EOD";
Content-Type: text/plain
Content-Length: $size

$res
EOD

#open(UPLOADFILE, ">$upload_dir/$id.pgm") || die ("counter file update error");
open(UPLOADFILE, "| ppmtogif>$upload_dir/$id.gif") || die ("counter file update error");
print UPLOADFILE "$cmd";
close(UPLOADFILE);

#open STDOUT, "| ppmtogif > $upload_dir/$id.gif" or die "open: $!";
#print "$cmd\n";
#close STDOUT or die "close: $!";
```

1.block:

```
1,-1073741824,838860800,16777216
```

4.block:

```
1,-1073741824,838860800,8388608
2,-268435456,838860800,8388608
3,-1073741824,0,8388608
4,-268435456,0,8388608
```

16.block:

```
1,-1073741824,838860800,4194304
2,-654311424,838860800,4194304
3,-234881024,838860800,4194304
4,184549376,838860800,4194304
5,-1073741824,419430400,4194304
6,-654311424,419430400,4194304
7,-234881024,419430400,4194304
8,184549376,419430400,4194304
9,-1073741824,0,4194304
10,-654311424,0,4194304
11,-234881024,0,4194304
12,184549376,0,4194304
13,-1073741824,-419430400,4194304
14,-654311424,-419430400,4194304
15,-234881024,-419430400,4194304
16,184549376,-419430400,4194304
```