1. On Configurability of Distributed Volunteer-Based Computing in the Comcute System

Paweł Czarnul

Gdansk University of Technology,
Faculty of Electronics, Telecommunication and Informatics,
Computer System Architecture Department
e-mail: pczarnul@eti.pg.gda.pl

Abstract

The chapter proposes additional solutions that can be implemented within the Comcute system to increase its configurability. This refers to configuration of the reliability level in the W and S server layers, static or on-the-fly data partitioning and integration, configuration of the system for processing in the data streaming fashion, extending the system for selection of a project that the client wants to contribute to, ease of migration of legacy codes to the system. Finally, an example of a legacy distributed application for monitoring client locations and resource usage is presented with suggestions on its migration to the Comcute system environment.

Keywords*: Comcute, distributed system configurability, volunteer-based computing*

1.1. Introduction

The Comcute system was designed [1-4] and implemented [5-9] as a flexible large-scale system for volunteer computing [10-11], resembling the well-known BOINC [12] initiative. Namely, it allows volunteers connecting to the system to fetch codes and subsequently input data packets and return results for the packets computed by the code to the server side. The system allows definition of several tasks and starting computations for several task instances at the same time. Compared to the BOINC system, Comcute puts more emphasis on reliability and dependability on the server side. The architecture of the system is depicted in Fig. 1.1 and distinguishes the following layers:

1. Z-layer – where the system client defines new tasks, starts instances of previously defined tasks, tracks statuses of running tasks and fetches results for completed tasks,
2. W-server layer – the servers supervising execution of tasks. By design, this layer is not accessible directly by clients running codes for data packets. For each task instance, a subset of W servers is arranged that partitions the task among its members. The servers are in charge of the execution. The tasks pass input data packets for the task instance to connected S servers beneath them as well as collect and merge results obtained from the S layer.

3. S-server – these are distribution servers that are exposed to clients who fetch execution code and subsequent data packets and return results for these data packets.
4. I-client level – this is an untrusted layer of regular clients fetching and returning results to the system.

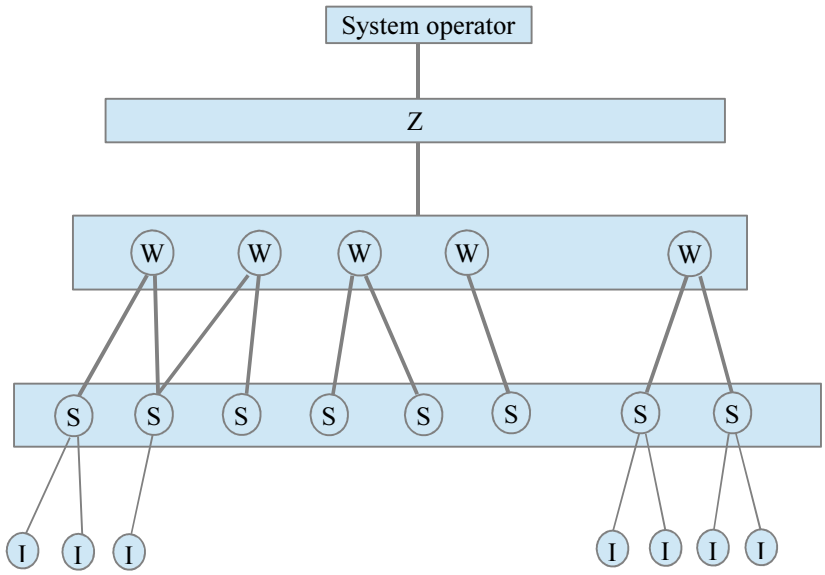

Fig. 1.1. Architecture of the Comcute System

1.2. Configurability

This section discusses the flexibility of the system in terms of several aspects:

1. configuration of the reliability level in the W and S server layers,
2. static or on-the-fly data partitioning and integration,
3. configuration of the system for processing in the data streaming fashion,
4. extending the system for selection of a project that the client wants to contribute to,
5. ease of migration of legacy codes to the system.

1.2.1. Distribution of data among W and S servers

The system architecture has been designed in such a way that allows tuning the trade-off between reliability and performance at two levels:

1. W servers – after a task has been defined and scheduled for execution, a subgroup W' of group W is created and parts of the initial data set are assigned to the members of the group. Note that data packets for a task should be distributed to more than one W server to allow for better geographical dispersion.

a. Additionally, if required, task execution could be launched in many copies at the W server level. In this case, more than one W' subgroup would be formed. At the cost of increased resource utilization, the reliability level would be increased as such copies would most likely be executed in very geographically distributed regions.

2. S servers – depending on the required redundancy level, each server could serve more copies of a data packet to incoming clients. However, this is hazardous from the reliability point of view as an active client connecting many times to the same S server could try to capture all copies of the same data packet and provide incorrect results for them. Assuming there is no other copy distributed to another S server, the system would not be able to detect such an attack and even identify the IP of the offender. From this point of view, copies of a data packet should be rather distributed among different W servers and underlying S servers connected to them. This gives a better possibility to distribute data packets geographically.

1.2.2. Static or on-the-fly data partitioning and result integration

The initial version of Comcute performs the following steps regarding task definition and execution:

1. definition of a task,
2. launching an instance of the task,
3. provision of input data for the task. This can also be done dynamically at runtime, as long as the task has not terminated. This allows the type of processing similar to data streaming,
4. data partitioning (before execution starts),
5. distribution of data packets to S servers, clients and collecting results the data packets at the W server level,
6. after all data packets have been processed by the clients (including the redundancy level requirement) launching data integration.
7. After data integration has been finished, store the final result in a database for the Comcute client to fetch it later.

While this approach works well for most applications, such as:

1. finding prime numbers within the given range (the size of output data is manageable),
2. finding potential solutions for yet unproved theorems such as Collatz's theorem (in this case output data appears only if input data matching the given theorem is found),
3. finding big numbers matching certain criteria such as big prime numbers, Mersenne's numbers etc.

it may not be adequate in all cases. Namely, the following constraints may appear:

1. data size limitations: the total size of output data for all data packets may be difficult to store at once.

2. the application may require partial data to show up as the application is still running. For example, the application for reporting subsequent locations and resource usage of the client computer [13] works in such a way that:
    a. each client I receives a data packet; its content is ignored,
    b. the client sends back its location and usage of the resources such as the processor and memory.

    In this case, it would be preferable to monitor the location of the client (and possibly the client leaving allowed space) as the application is running (there are data packets) rather than after all the packets have been processed.

To overcome these limitations, there are several solutions possible:

1. partitioning of the initial data set into disjoint data sets on which several instances of a task would be based. In this case, effectively the initial task is partitioned into several subtasks. This solution has a drawback that it needs to be done manually with the knowledge of how fast particular data packets for the given task can be processed.
2. partitioning and merging on-the-fly.

In the latter case, the algorithm can work as follows:

1. definition of a task,
2. launching an instance of the task,
3. provision of input data for the task. This can also be done dynamically at runtime, as long as the task has not been terminated. This allows the type of processing similar to data streaming,
4. data partitioning up to the desired and defined number of data packets (before execution starts). After the given number of data packets has been generated, the packets are passed to W and subsequently S servers and the clients. In the meantime, generation of a next batch of data packets proceeds effectively overlapping data generation and communication. This also speeds up the start of processing by the end users. Finally, it allows to have precise control over the data size footprint for the given application. This means, that based on task priorities, more or less data space during execution could be granted. Consequently, based on these settings allowing a larger or fewer number of applications running concurrently and in parallel could be possible. It is clear that this change clearly contributes to increasing the processing flexibility for not only a single but all applications running at the given moment.
5. distribution of data packets to S servers, clients and collecting results the data packets at the W server level,
6. after a certain number of data packets have been processed by the clients (including the redundancy level requirement) or after a certain time limit since the last data integration moment has elapsed – launch data integration. In this case, there must be a global state of the already integrated data packets to which results of new data packets will be added.
7. After data integration has been finished, store the final result in a database for the Comcute client to fetch it later.

Additionally, merging partial results as the application is running may influence new data packets being generated. For instance, results of analysis of a particular data range may result in cutting off another data range in which it is known at this point in time that no solution will be found. This can also reduce the total running time of the task considerably.

This brings the question related to implementation – namely how dynamic partitioning and merging will be performed:

1. partitioning: the question arises how the state of the already integrated results should be stored. Since integration is performed by a partitioner object then the data itself can be stored in a database or file(s) with proper references from the partitioner itself holding control variables denoting where the integration finished. In this case, the size of the partitioner object can be kept small and the partitioner object can be serialized and deserialized when it is needed. Copies of such footprints should also be saved in case of server failures. Alternatively, such control variables can also be stored in a database or files in which case the partitioner object should be able to restart from the state saved there.
2. merging: analogous to the partitioning process regarding saving the state of the data which should be stored in a database/files. The same applies to retrieving control variables/information about the state.

1.2.3. Data streaming

While originally processing in the system has been designed to process input data files uploaded before simulation, it can be extended to support data streaming. This can be accomplished in two ways without rewriting the internals of the system in the W server layer:

1. Currently, until computations are still running, it is possible to upload new data packets that will be processed as they arrive. Consequently, it would be possible to install a proxy that would periodically receive data packets from an external source and would forward to the Comcute system. This would, however, require an API to do so for a particular task instance. The following API would suffice along with login/password information:

   ```
   newDataSubmission(taskInstanceId, loginCredentials, data);
   ```

2. It is possible to split the initial data externally into several tasks and correspondingly run more task instances as new input data becomes available.

Additionally, especially for scenario 1 above, it should be defined when processing of data streaming should end. Two possibilities can be implemented here:

1. the client externally marks the task instance as finished indicating it will no longer accept new data packets and will change its state to “COMPLETED” after all current data packets have been processed,
2. in the task instance definition it will be denoted what will be the total expected data size passed to the task instance after which it will change its state to “COMPLETED”.

1.2.4. Project selection

Currently, the system works in such a way that tasks defined at the Z and W server layers are completely transparent to clients connecting to the system and wishing to contribute to computations. This stems from the assumed design principle that the W server layer should be completely invisible to clients and the S server layer should act as the distribution layer only and should know nothing about the content of the data forwarded from W to I and vice versa.

From the point of view of the client, however, it may be preferable to know to which task/project the client is contributing or even select the task. This is possible but would need the following extensions in the system internals - extending:

1. the knowledge of the S servers to know about data packets and association of data packets with particular tasks,
2. the initiation phase so that the client fetches a list of available tasks and information about these. Two alternative solutions are now possible:
    a. the client (upon each connection) sends an identifier of the task it is interested in and the S server responds by sending a data packet for this particular task.
    b. the client sends back its interest in performing computations for a particular task to the S server. The S server then keeps a map of recently connected clients I and tasks to which the clients have subscribed. As soon as a client connects to the S server, it needs to introduce itself and the S server sends back an appropriate data packet. However, this requires the need for the client to introduce itself which may be a problem for a client running within a web browser (such as unsigned applet). For this reason, solution 1 above may be preferable.

1.2.5. Code migration to volunteer based processing

The current processing paradigm in the Comcute system resembles the well-known master-slave [14] or map-reduce [15] paradigm in which input data is divided among slave nodes and then merged (reduced) by the master. In Comcute, it is internally distributed further as more W and S servers are involved in parallel solving of the same task instance using distinct data packets.

This paradigm is then reflected in the API for the Comcute programmer, namely the need for provision of:

1. partitioner,
2. linker,
3. computational code.

Each master-slave program can be naturally mapped to this paradigm requiring just translation of the original code into the code used in the Comcute system (such as Java or JavaScript).

It is interesting, though, how other popular parallel processing paradigms [14] could be mapped to the Comcute system:

1. SPMD – Single Program Multiple Data in which neighbors do need to exchange boundary information among one another between iterations of the main processing loop. Unfortunately, due to dense communication in this case and comparable communication times and computational times (of a single iteration) even on a cluster, it is not possible to effectively parallelize it in the Comcute system. For instance, computing of a single iteration of a simulation on a modern processor can take in the order of a fraction of a second up to 1-2 seconds which can be much shorter than the communication time with the server. However, it is possible to approach the problem from a different perspective i.e. launch complete simulations with different input data sets on different client computers. As an example, a simulation for fire spreading was developed for the Comcute system. In each simulation, a different set of locations for fire stations is examined in the context of fire spreading. Each such simulation is launched separately on a different client computer while the Comcute system integrates results for those simulations. Precisely, each simulation results in information how effective the locations of the fire stations were. Finally, the Comcute system can determine the best locations of planned fire stations assuming potential various places where fire starts, various wind conditions etc.

   Thus, in the context of migration of SPMD simulations to the Comcute system, this would require translation of the simulation code to one of the languages used by the client side in the Comcute system (such as Java, JavaScript) and proper definition of input data so that various data sets could be generated when partitioning the initial data. For instance, for the aforementioned fire spreading simulation, various data packets with various locations could be generated.

2. Pipelining – similarly to SPMD this scheme is not well suited to running in the Comcute system unless:

   a. communication times are much smaller than times of computations in successive stages of the pipeline,

   b. clients do have specialized hardware that makes processing of various stages much faster (such as specialized GPU or other devices). This, however, would require further extension of the system with more knowledge about the client. In particular, the client would need to pass to the server not only the information what technologies it can use but also how effective its hardware for particular types of computations is.

   It should be noted that, as in the SPMD case, legacy pipelining code could be migrated to the Comcute system to perform distinct simulations using various input data sets.

3. Divide-and-conquer – this paradigm is much more promising than the previous two. First, it is naturally expanded into a tree of subproblems for which results need to be then integrated in the top part of the tree. This means that the top of the divide-and-conquer tree in the initial phase can be migrated to the partitioner code, lower part of the tree to computational code and the top of the divide-and-conquer tree in the final phase can be migrated to the linker code. Partitioning, linking and computations must be provided in the divide-and-conquer computations anyway so it comes down to running proper

translators and minor modifications of the code to run within the partitioner, linker and client framework.

Additionally, as in client-server frameworks, it may be desirable to provide a communication framework for encapsulation of data passed between the partitioner, computational code and the linker for the various languages supported on the client side in the Comcute system. Namely, it could assist in serializing/deserializing data that is passed between these components.

1.3. Example – an application for monitoring locations and resource usage of client computers

Following the discussion on the configurability presented in previous sections, it is now discussed how to implement and configure a real application that is aimed at monitoring locations and resource usage of clients connecting in the Comcute system. Rather than asking each client to perform computations for a data packet, in this case each data packet is treated as kind of a request from the server side to the client to respond with its location and resource usage statistics. From this point of view, it is a different application from the mainstream Comcute examples.

1.3.1. Functional specification

Functional specification of the application is depicted in Fig. 1.2. where usecases from the operator and client points of views are presented.

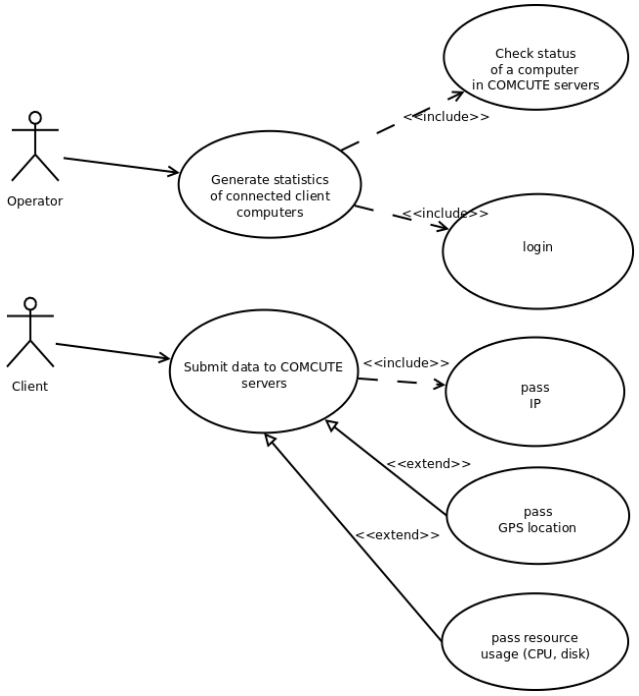

Fig. 1.2. Application usecases

1.3.2. Design and implementation of legacy code

Fig. 1.3 depicts an interaction diagram for communication between the client and the system with registration of new location and usages (operation register of a Web Service) and monitoring (operations checkifmoved() and checkusage()) as designed in [13].

The author has implemented the solution which is based on three cooperating applications:

1. ComcuteClientCheckerJavaApplication – an application for the system operator that allows him/her to monitor locations and states of resources connected to the system.
2. ComcuteClientMonitorClientJavaApplication – an application run by the client that allows registration of the client location and resources usage as well as the IP of the client.
3. ComcuteClientMonitorEJBModule1 – a server side application allowing clients to invoke Web Services to submit information about location and resource usage.

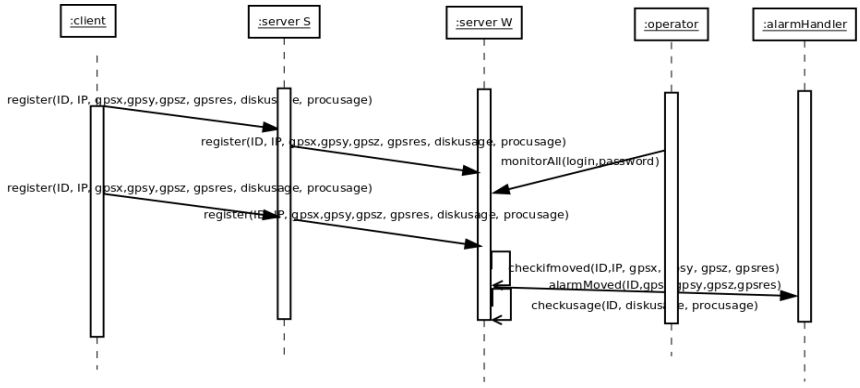

Fig. 1.3. Interaction diagram for the application

ComcuteClientCheckerJavaApplication

The main part of the application is depicted in Lst. 1.1. This Java client application invokes operation monitorAll of a Web Service on the server side. This allows an operator (who needs to provide a login and a password) to monitor clients connected to the system. The application displays information about locations of the clients along with information about clients leaving the desired and allowed space and resource usage. The application calls the server every 5 seconds.

Lst. 1.1. ComcuteClientCheckerJavaApplication source code

```
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
```

```
        // check the status
        String status;

        try {
       for(int i=0;i<100;i++) {
          status=monitorAll("login","password");
          System.out.println(status);
          Thread.sleep(5000);
        }
        } catch (java.lang.InterruptedException e) {

     ....
        }
    }

    private static String monitorAll(java.lang.String login, java.lang.String password)
    {
      clientmonitor.ClientMonitorWebServiceService service
            = new clientmonitor.ClientMonitorWebServiceService();
       clientmonitor.ClientMonitorWebService port = service.getClientMonitorWebServicePort();
       return port.monitorAll(login, password);
    }
}
```

ComcuteClientMonitorClientJavaApplication

Application ComcuteClientMonitorJavaApplication allows each client to submit information about location, IP address as well as usage of processor and disk space. Each client is identified by a MAC address. The application fetches the client IP address which passes along with GPS location and resource usage. Approximate location can also be determined based on the IP address [17,18]. Main parts of the application are shown in Lst. 1.2.

Lst. 1.2. ComcuteClientMonitorClientJavaApplication source code

```
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        // first get the address of this Comcute client

   byte mac []=null;
   String IP=null;
        try
          {
     Enumeration eth = NetworkInterface.getNetworkInterfaces();

     while (eth.hasMoreElements())
     {
       NetworkInterface eth0 = (NetworkInterface) eth.nextElement();

       mac = eth0.getHardwareAddress();
```

```
       // check if the mac is not null
       if (!(mac == null)) break;

      }
      // get the IP
      IP =InetAddress.getLocalHost().getHostAddress();
        }
    catch (Exception e)
    {
      e.printStackTrace();
    }


if (mac!=null) {

    StringBuilder sb=new StringBuilder();
    for (int k = 0; k < mac.length; k++) {// convert the address to a string
              sb.append(String.format("%02X%s", mac[k], (k < mac.length - 1) ? "-" : ""));
    }

    String macString=sb.toString();
    System.out.println("Found MAC address of my computer: "+macString);
    System.out.print("This address will be used as an id when passing information about location
and resource usage.");
    // now register a location in the Web service
    Random randomGenerator = new Random();
    double currentX,currentY,currentZ;
    try {
      for (int i=0;i<100;i++) {
        currentX=100*randomGenerator.nextDouble();
        currentY=100*randomGenerator.nextDouble();
        currentZ=10*randomGenerator.nextDouble();

        System.out.println("Sending my current location
                  ("+currentX+","+currentY+","+currentZ+" to Comcute");
      register4(macString,IP,50.0,50.0,currentX,currentY,currentZ);
      Thread.sleep(5000);

         }

        }  catch (java.lang.InterruptedException e) {

        e.printStackTrace();
        }
      }
    }

    private static String register4(java.lang.String macAddress, java.lang.String ip, double
            diskUsage, double cpuUsage, double gpsx, double gpsy, double gpsz) {
        clientmonitor.ClientMonitorWebServiceService service = new
            clientmonitor.ClientMonitorWebServiceService();
        clientmonitor.ClientMonitorWebService port = service.getClientMonitorWebServicePort();
        return port.register4(macAddress, ip, diskUsage, cpuUsage, gpsx, gpsy, gpsz);
    }
}
```

ComcuteClientMonitorEJBModule1

The server-side ComcuteClientMonitorEJBModule1 application was developed as Web Services deployed in GlassFish. For connected clients it stores information such as IP addresses, locations and resource usage.

Operations register* allow submission of information while operations monitor* allow fetching information (after provision of login and password) along with information about clients leaving desired locations. Main parts of the application are shown in Lst. 1.3.

Lst. 1.3. ComcuteClientMonitorEJBModule1 source code

```
@WebService()
@Stateless()
public class ClientMonitorWebService {

    static HashMap currentLocations=new HashMap();
    static HashMap diskUsages=new HashMap();
    static HashMap cpuUsages=new HashMap();
    static HashMap IPs=new HashMap();




    /**
     * Web service operation
     */
    @WebMethod(operationName = "monitorAll")
    public String monitorAll(@WebParam(name = "login")
    String login, @WebParam(name = "password")
    String password) {

        // monitor all locations if any is outside of where it should be

        StringBuilder retVal=new StringBuilder();

        retVal.append("Statuses of clients:");

        Iterator iterator = currentLocations.keySet().iterator();

        while(iterator. hasNext()){
            String macAddress=(String)iterator.next();
            GPSLocation clientLocation=(GPSLocation)currentLocations.get(macAddress);

            if ((clientLocation.gpsx>90) || (clientLocation.gpsy>90) || (clientLocation.gpsz>8)) {
                retVal.append(" Client "+macAddress+" outside of allowed space : ("+
clientLocation.gpsx+","+clientLocation.gpsy+","+clientLocation.gpsz);

            } else
                retVal.append(" Client "+macAddress+" in allowed space : ("+
clientLocation.gpsx+","+clientLocation.gpsy+","+clientLocation.gpsz);

            String IP=(String)IPs.get(macAddress);
            String diskUsage=((Double)diskUsages.get(macAddress)).toString();
            String cpuUsage=((Double)cpuUsages.get(macAddress)).toString();

            retVal.append(" IP="+IP+" diskUsage="+diskUsage+" CPUUsage"+cpuUsage);
        }
```

```
        return retVal.toString();

    }
    /**
     * Web service operation
     */
    @WebMethod(operationName = "register_4")
    @RequestWrapper(className = "clientmonitor.register_4")
    @ResponseWrapper(className = "clientmonitor.register_4Response")
    public String register(@WebParam(name = "macAddress")
    String macAddress, @WebParam(name = "IP")
    String IP, @WebParam(name = "diskUsage")
    double diskUsage, @WebParam(name = "cpuUsage")
    double cpuUsage, @WebParam(name = "gpsx")
    double gpsx, @WebParam(name = "gpsy")
    double gpsy, @WebParam(name = "gpsz")
    double gpsz) {

        // add the location of the client to the hash map
        currentLocations.put(new String(macAddress), new GPSLocation(gpsx,gpsy,gpsz));

        // add the IP and the usages
        IPs.put(new String(macAddress), new String(IP));

        diskUsages.put(new String(macAddress), new Double(diskUsage));
        cpuUsages.put(new String(macAddress), new Double(cpuUsage));

        return null;
    }

}
```

1.3.3. Running the application

The following figures show screenshots of the working application run within the Netbeans environment. Applications run in the following order:

- ComcuteClientMonitorEJBModule1,
- ComcuteClientMonitorClientJavaApplication (at least one),
- ComcuteClientCheckerJavaApplication.

Each client submits its data (location and resource usage) every 5 seconds and a system operator monitors states of the clients, in particular clients leaving the desired space.

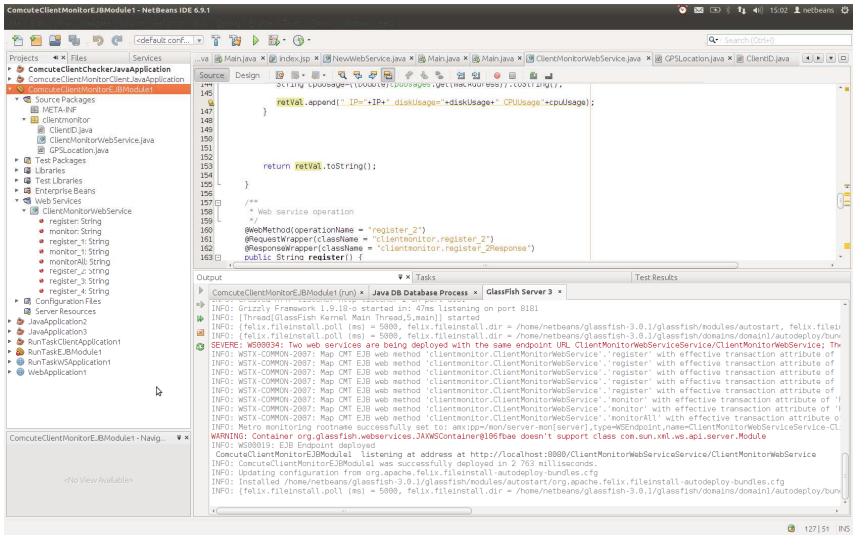

Fig. 1.4. Screenshot - deployment

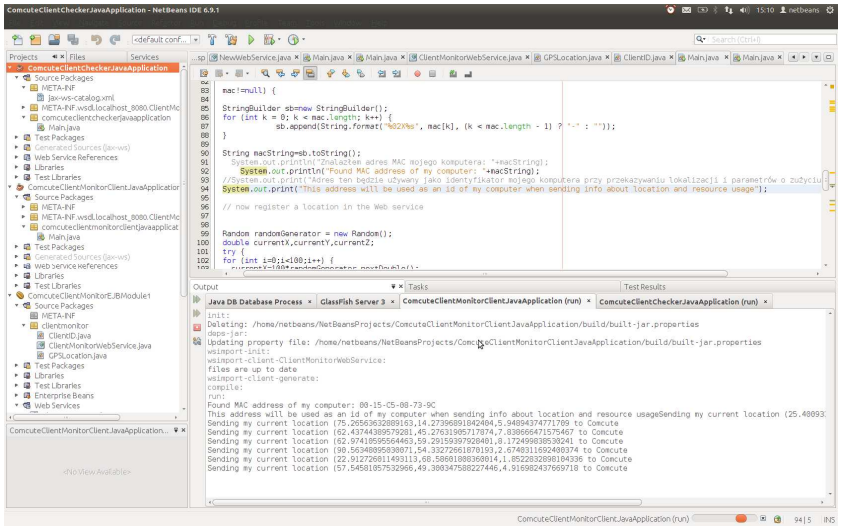

Fig. 1.5. Screenshot – sending data to server

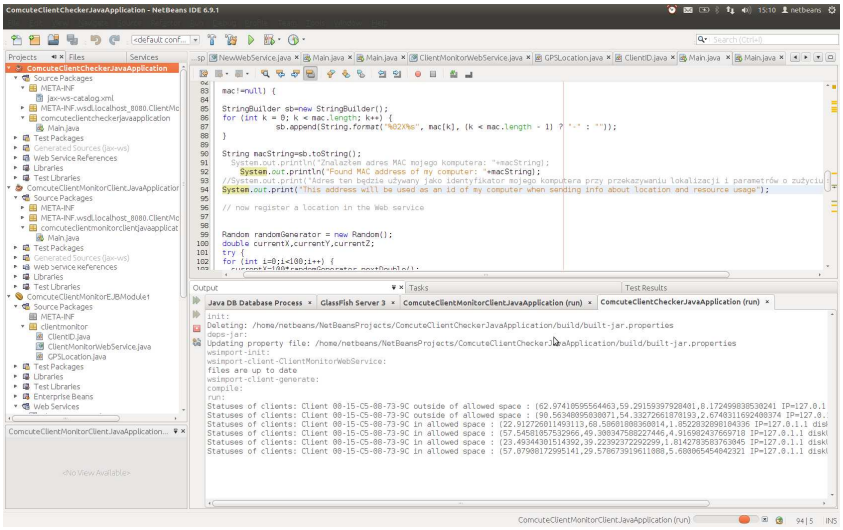

Fig. 1.6. Screenshot – fetching client data

1.4. Migration and configurability in the Comcute environment

We now discuss how to migrate and configure an application in the Comcute environment.

It should be noted that a client application must be run in such a way that a unique identifier is possible to fetch (such as the MAC address [16]). Furthermore, even though reporting data is quick and does not need time, each client should introduce intervals (such as the 5 seconds or more in the testbed application) to prevent from flooding the server with requests. Additionally, in Comcute, each data packet would correspond to one client reporting one location. Assuming there are N clients reporting its location every INTERVAL seconds then for time T, N*(T/INTERVAL) data packets would be needed. As long as the application is running, new data packets can be provided to extend the application working time.

Furthermore, proper integration of results would be required on the server side. Namely, successive locations of a client should be stored (either one location per client or a history of locations) as soon as available to find out the client leaving the allowed space as soon as possible. Thus, rather than defining a certain number of data packets after which integration is performed, an interval should be used. In the former case, if intervals between data packet processing are large, so will be the interval between integration actions. In the latter case, it is defined *a priori* without flooding the server as well.

References

1. J. Balicki, J. Kuchta,, M. Matuszek, P. Czarnul, P. Szpryngier, P. Brudło. *Functional design of Comcute, Technical report no* 33/2011, in Polish.
2. J. Kuchta, M. Matuszek, P. Czarnul, P. Szpryngier. *Design of architecture of the Comcute laboratory equipment, Technical report WETI no* 32/2011, in Polish.

3. J. Balicki, M. Matuszek. *Requirements analysis for simulations of selected parameter configuration in Comcute, Technical report WETI no* 35/2011, in Polish.
4. J. Kuchta, P. Szpryngier, J. Szymański, P. Brudło. *Hints for implementation of selected computational methods for Comcute, Technical report WETI no* 36/2011, in Polish.
5. T. Bieliński, Ł. Gadomski, M. Nocula: Report on Java implementation of a Comcute prototype, *Technical report WETI no* 38/2011, in Polish.
6. J. Balicki, T. Boiński. *Report on Comcute installation, Technical report WETI no* 39/2011, in Polish.
7. Polak. *ASP implementation of Comcute*, *Technical report WETI no* 37/2011, in Polish.
8. J. Kuchta, M. Matuszek, P. Czarnul. P. Szpryngier. *Requirements for implementation of Comcute, Technical report WETI no* 34/2011, in Polish.
9. J. Balicki. *Optimization of load balancing in Comcute*, *Technical report WETI no* 40/2011, in Polish.
10. O. Nov, D. Anderson, O. Arazy. *Volunteer Computing: A Model of the Factors Determining Contribution to Community-based Scientific Research.* WWW 2010. Raleigh, USA.
11. David P. Anderson, Carl Christensen, and Bruce Allen. *Designing a runtime system for volunteer computing*. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC '06). ACM, New York, NY, USA, , Article 126 . DOI=10.1145/1188455.1188586 http://doi.acm.org/10.1145/1188455.1188586
12. BOINC. home page http://boinc.berkeley.edu
13. P. Czarnul. *A distributed application for tracking client locations and monitoring client computers*, pp. 184-193 in Distributed computations in grid architecture systems, J. Balicki, J. Kuchta eds., 2012, in Polish.
14. Wilkinson, M. Allen. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers. Prentice Hall. 1998.
15. Apache Hadoop™ MapReduce, 2007, http://hadoop.apache.org/mapreduce/
16. MAC Address Java Applet, http://techdetails.agwego.com/2008/02/11/37/, February 2008
17. IP Address Location, 2012, http://www.ipaddresslocation.org/find-mac-address.php
18. S. Plexus. Updated Location Applet, http://iwtf.net/2011/04/23/updated-location-applet/