Net-Computing 2017
Efficiency in the buildings of Datacenters

Project Group 03
Rick de Jonge (s2775832)
Jan Boonstra (s2774100)
Remco van Buijtenen (s2714086)

April 3, 2017

1 Context and purpose

The purpose of our project is to develop an app that uses RMI, sockets, a RESTfull web interface and a message queue. In order for it to fit within the theme *"efficiency in datacenters"* we decided to have each computer in the network (in our "datacenter") measure it's CPU and RAM statistics during run-time. This data is then gathered by the server in a message queue where it is used to distribute tasks among the computers in the network in an attempt to optimize power consumption. These tasks can be given to the server through a web interface, which is currently limited to the compilation of c programs through the *gcc* command (when installed), and the execution of any linux executable file or windows *.exe* file generated by the gcc compiler. The server distributes this tasks by invoking remote methods on one of the computers in the network. When a task finishes executing this client computer then sends a message to the server, which in turn invokes a remote method to retrieve the output of the process. While a process is executing it can be viewed and interrupted by users through the web interface. After execution its command, execution times, exit status, standard output, standard error and standard in can be viewed.

2 User instructions

To run this project, you will need the two jar files with the server and worker code and a sigar file, specific to the operating system. The server can be run from the terminal using *java -jar server.jar <serverport>*, the worker needs as its arguments one of the following: *java -jar worker.jar <serverport>* (in this case the serveraddress will be *localhost*), *java -jar worker.jar <serveraddress> <serverport>* (both of these will use rmiport 1099) or *java -jar worker.jar <serveraddress> <serverport> <rmiport>*. The serveraddress is the ipaddress of the computer hosting the server (format: xxx.xxx.xxx.xxx), *localhost* is also a valid serveraddress. The serverport must be equal on both server and worker.
When the server and at least one worker are running, you can open the servers interface via your web browser, by going to *<server ip-address>:8080/task/*. In the menu you can select "Create a new task" to open a form where you can enter a command and its input. After you have done this, you can go to "active tasks" to view a list of the commands that you are running and that have yet to finish. If you go to "finished tasks", you will see a list of all your commands that have already finished. By clicking on the view button for one of the finished tasks you can view its input and output or, if something went wrong, its errors.

3 Design

The project will consist of two separate programs, namely the server and the worker programs. The server will be connected to all the workers, collecting information about them and deciding where tasks will be executed. The workers will monitor their cpu usage and their available memory and have a remote method invocation server to be able to execute tasks. We have limited the tasks to be console commands, because they are a simple representation of more complicated programs, but still show us what the application should do.

3.1 Program language

The primary language for the project is Java, the server and the workers are created as different java projects. Both of these projects include resources located in a shared project.

3.2 Components

3.2.1 Web interface

The user will interact with the web interface. On this webpage, one will be able to send a new task to the server for execution. The send tasks are displayed and sorted in two categories, namely unfinished and finished. The user can request more information about finished tasks, including the output of the program.

- Webserver.java
  The webserver is the Server's main class. It starts all components needed for receiving messages, scheduling tasks and it starts a Jetty web-server
- HTMLTemplate.java and TaskApi.java
  These classes retrieve the requests of the users from the webservice using http-requests and add those to the task queue, ready to be send to a worker.

3.2.2 Server

The server is the main connection between the user and the workers. The server retrieves any tasks from the webservice using http requests. It contains an inbox that will store unhandled tasks and the status of the tasks, and a scheduler that will assign the tasks it receives to a worker, using remote method invocation to execute the tasks.

- DataAnalyzer.java
  Looks through the *MessageQueue* for messages with messagetype 1(*Measurement*) or 2 (*TaskInfo*), if there is a message with a messagetype 1, the data analyzer will dequeue the message and update the connection data. If there is one with messagetype 2, the the output of the program will be retrieved using the RMI server.
- Connection.java
  The connection class holds all information the server knows about a worker. This includes the initial data, the last measurement and the connection.
- ConnectionList.java
  This list holds all connections of the server. The list is kept sorted, in order to retrieve the next worker for a next task as fast as possible.
- Message.java
  A message is a general way to store server requests from either the webservice or a worker. It holds a message, which can be a task or measurement, and a connection.
- MessageInbox.java
  This thread runs has an opened socket, listening for more workers to make a connection. If it receives one, it creates a new *MessageReceiver* and adds the connection to the *ConnectionList*.
- MessageQueue.java
  This extention of a linked list has an extra function to be able to peek at the contents of a message.
- MessageReceiver.java
  This is the connection between the server and a worker, it remains opened and is able to receive messages to put into the *MessageInbox*.

- Task.java, TaskActive.java and TaskFinished.java
  This structure of tasks are created such that a task gets more information attached when it is active (the process id and starting time) and even more when it finishes (the output, including errors in execution).
- TaskQueue
  This queue contains the tasks that still have to be assigned to a worker.
- TaskList
  This is the list with tasks that are assigned to a worker and therefore have either the active or finished status.

3.2.3 Workers

Servers will regularly send their information, like available space, to the central controller. Workers are basic units, and can be easily multiplied in order to extend the network.

- Worker.java
  This class first collects the server address and port, and the port for the RMI server from the arguments and then starts a *ResourceMonitor* and binds the RMI server.
- ResourceMonitor.java
  The *ResourceMonitor* is a thread that creates a *Measurement* objects with information about the machine. It sends this data periodically to the *MessageInbox*.
- Task.java
  This is the workerside representation of a task, the objects holds information like task id, input, output and errors. The task is executed in a running thread.
- TaskManager.java
  This class manages all the tasks that are running on the specific worker, it can stop tasks, collect the output and notify the server if a task finishes.

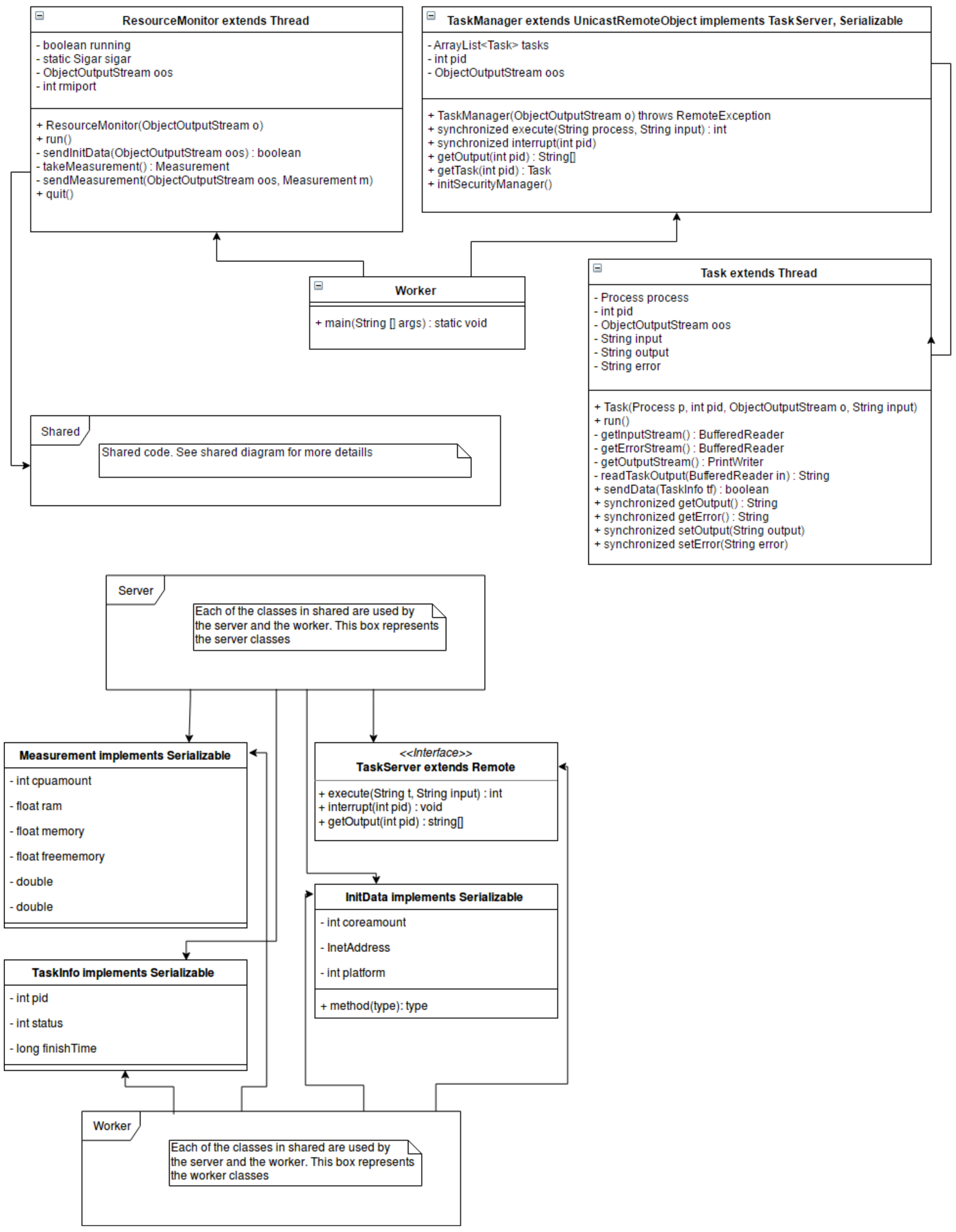
3.2.4 Shared code

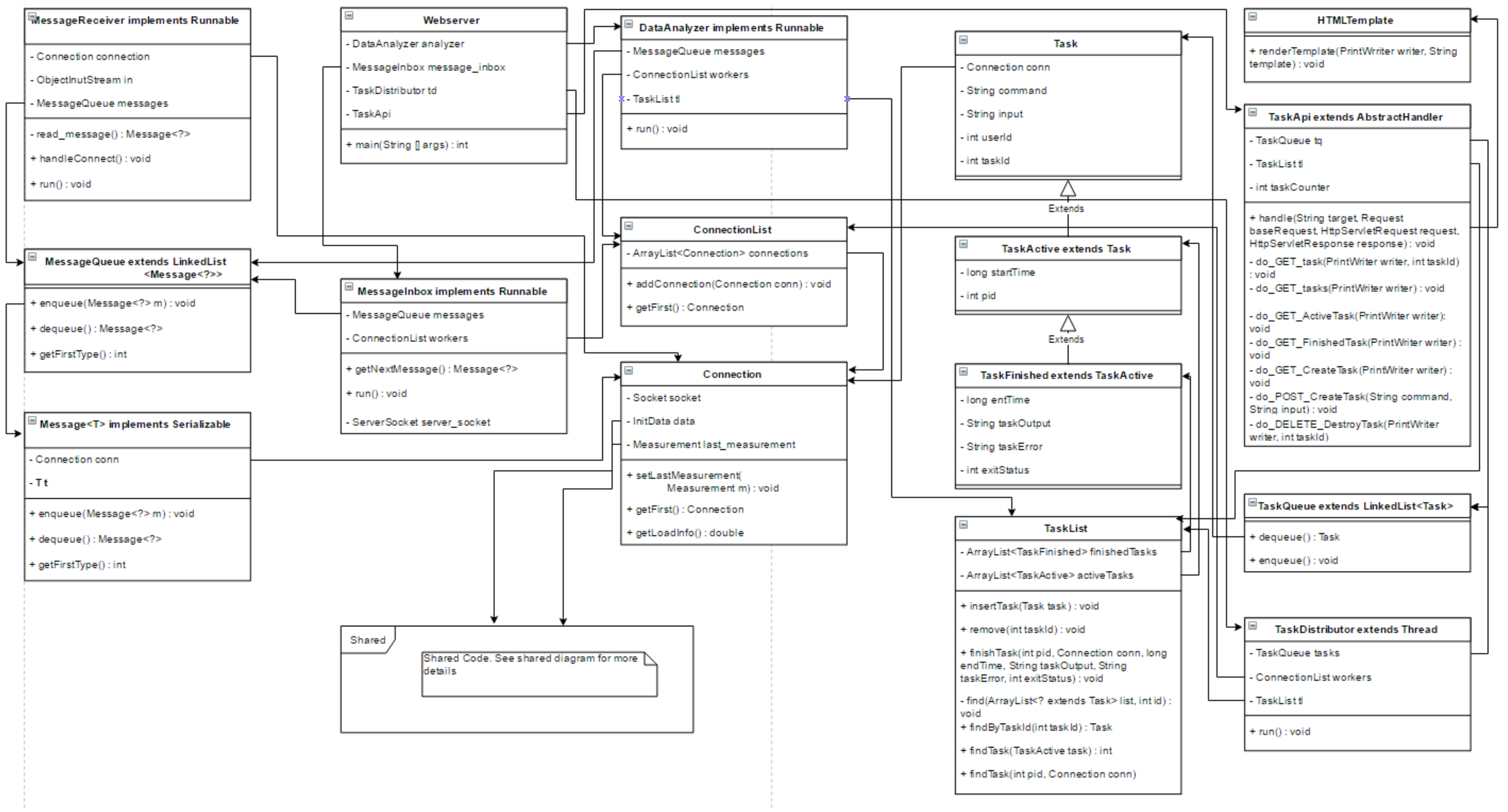
Since the server and workers will be communicating with each other, the two projects will have shared classes. These data classes and shared libraries are stored in a separate java project.

- InitData.java
  The *InitData* class is the data that is send when a worker first connects to the server, giving its basic information like the ip address, platform and the amount of cores.
- Measurement.java
  This data class is send periodically by the worker, sending information about how much of the computing power of the worker is used.
- TaskInfo.java
  This data item is the representation of a task that the server got from the user, while it is running on one of the workers.
- Shared libraries
  The shared code also contains the libraries needed to run the *sigar*[1] methods, for windows and linux.

[1]Sigar is a library we found that will measure machine data for us. We use it to collect the CPU and RAM usage

4 Diagrams


ResourceMonitor extends Thread
- boolean running
- static Sigar sigar
- ObjectOutputStream oos
- int rmiport
+ ResourceMonitor(ObjectOutputStream o)
+ run()
- sendInitData(ObjectOutputStream oos) : boolean
- takeMeasurement() : Measurement
- sendMeasurement(ObjectOutputStream oos, Measurement m)
+ quit()
TaskManager extends UnicastRemoteObject implements TaskServer, Serializable
- ArrayList<Task> tasks
- int pid
- ObjectOutputStream oos
+ TaskManager(ObjectOutputStream o) throws RemoteException
+ synchronized execute(String process, String input) : int
+ synchronized interrupt(int pid)
+ getOutput(int pid) : String[]
+ getTask(int pid) : Task
+ initSecurityManager()
Worker
+ main(String [] args) : static void
Task extends Thread
- Process process
- int pid
- ObjectOutputStream oos
- String input
- String output
- String error
+ Task(Process p, int pid, ObjectOutputStream o, String input)
+ run()
- getInputStream() : BufferedReader
- getErrorStream() : BufferedReader
- getOutputStream() : PrintWriter
- readTaskOutput(BufferedReader in) : String
+ sendData(TaskInfo tf) : boolean
+ synchronized getOutput() : String
+ synchronized getError() : String
+ synchronized setOutput(String output)
+ synchronized setError(String error)
Shared
Shared code. See shared diagram for more detaills
Server
Each of the classes in shared are used by the server and the worker. This box represents the server classes
Measurement implements Serializable
- int cpuamount
- float ram
- float memory
- float freememory
- double
- double
<<Interface>>
TaskServer extends Remote
+ execute(String t, String input) : int
+ interrupt(int pid) : void
+ getOutput(int pid) : string[]
InitData implements Serializable
- int coreamount
- InetAddress
- int platform
+ method(type): type
TaskInfo implements Serializable
- int pid
- int status
- long finishTime
Worker
Each of the classes in shared are used by the server and the worker. This box represents the worker classes


MessageReceiver implements Runnable
- Connection connection
- ObjectInutStream in
- MessageQueue messages
- read_message() : Message<?>
+ handleConnect() : void
+ run() : void
Webserver
- DataAnalyzer analyzer
- MessageInbox message_inbox
- TaskDistributor td
- TaskApi
+ main(String [] args) : int
DataAnalyzer implements Runnable
- MessageQueue messages
- ConnectionList workers
- TaskList tl
+ run() : void
Task
- Connection conn
- String command
- String input
- int userId
- int taskId
HTMLTemplate
+ renderTemplate(PrintWrriter writer, String template) : void
TaskApi extends AbstractHandler
- TaskQueue tq
- TaskList tl
- int taskCounter
+ handle(String target, Request baseRequest, HttpServletRequest request, HttpServletResponse response) : void
- do_GET_task(PrintWriter writer, int taskId) : void
- do_GET_tasks(PrintWriter writer) : void
- do_GET_ActiveTask(PrintWriter writer): void
- do_GET_FinishedTask(PrintWriter writer) : void
- do_GET_CreateTask(PrintWriter writer) : void
- do_POST_CreateTask(String command, String input) : void
- do_DELETE_DestroyTask(PrintWriter writer, int taskId)
Extends
ConnectionList
- ArrayList<Connection> connections
+ addConnection(Connection conn) : void
+ getFirst() : Connection
TaskActive extends Task
- long startTime
- int pid
MessageQueue extends LinkedList <Message<?>>
+ enqueue(Message<?> m) : void
+ dequeue() : Message<?>
+ getFirstType() : int
MessageInbox implements Runnable
- MessageQueue messages
- ConnectionList workers
+ getNextMessage() : Message<?>
+ run() : void
- ServerSocket server_socket
Extends
Connection
- Socket socket
- InitData data
- Measurement last_measurement
+ setLastMeasurement(Measurement m) : void
+ getFirst() : Connection
+ getLoadInfo() : double
TaskFinished extends TaskActive
- long entTime
- String taskOutput
- String taskError
- int exitStatus
Message<T> implements Serializable
- Connection conn
- T t
+ enqueue(Message<?> m) : void
+ dequeue() : Message<?>
+ getFirstType() : int
TaskQueue extends LinkedList<Task>
+ dequeue() : Task
+ enqueue() : void
TaskList
- ArrayList<TaskFinished> finishedTasks
- ArrayList<TaskActive> activeTasks
+ insertTask(Task task) : void
+ remove(int taskId) : void
+ finishTask(int pid, Connection conn, long endTime, String taskOutput, String taskError, int exitStatus) : void
- find(ArrayList<? extends Task> list, int id) : void
+ findByTaskId(int taskId) : Task
+ findTask(TaskActive task) : int
+ findTask(int pid, Connection conn)
TaskDistributor extends Thread
- TaskQueue tasks
- ConnectionList workers
- TaskList tl
+ run() : void
Shared
Shared Code. See shared diagram for more details