ZetaGrid – Grid Computing Engine for Loosely Coupled Work Units

Sebastian Wedeniwski

Release notes

The ZetaGrid version 1.9.3-alpha is just a pre-release which cannot be used in a production environment. Version 1.9.3 is the first release which is published at IBM alphaWorks (at http://www.alphaworks.ibm.com) under IBM's International License Agreement for Non-Warranted Programs. The version 1.9.2 is the last stable code which is available at http://www.zetagrid.net only. But the version 1.9.2 will be deprecated and removed when version 1.9.3 is final.

The version 1.9.3-alpha contains some new features which are not completed and tested:

- A program as a part of a task can be specified for a specific number of used processors.
- A resource provider does not need to trust all programs of a task.
- The package structure is completely restructured.
- The API is simplified.

This document is just a first DRAFT and is not complete.

Overview

ZetaGrid is a grid computing platform designed for and specialized in solving very large and computing intensive problems in a heterogeneous and dynamic environment. The interactions, performance, availability and scalability of this technology is proven in multiple administrative domains, involving more than 10,000 computers in a heterogeneous and dynamic environment. The first project of ZetaGrid was a mathematical one -- the verification of the zeros of the zeta function. That's where the name comes from. ZetaGrid was initially developed and tested in the IBM-Lab Boeblingen. Further successful projects are image deconvolution in Life Science and a Boolean satisfaction (SAT) solver in Computer Science.

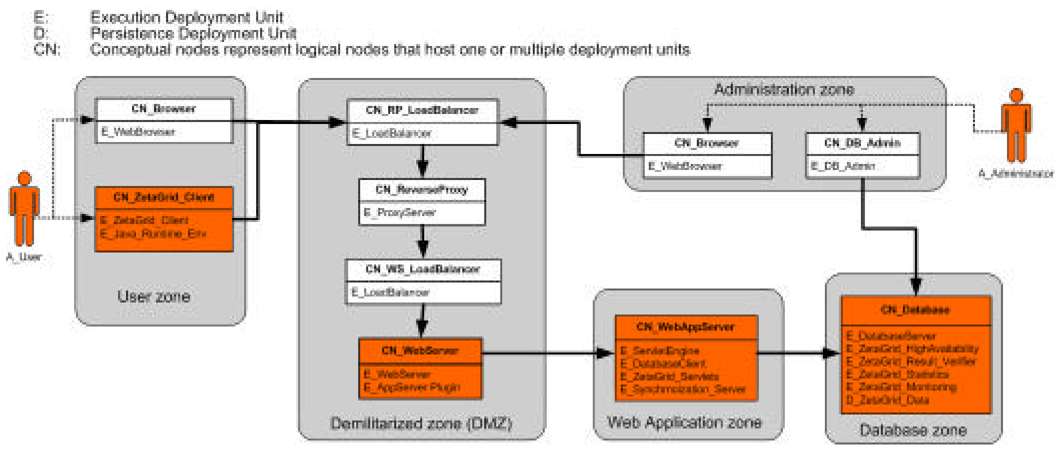
The goal of ZetaGrid is to use free capacity of voluntarily participating cross platform computers (even game consoles or mainframes) that can use slow communications. One of the most important factors for the acceptance of such a computational grid system is to ensure privacy and control for the participating clients. This platform schedules a configured large number of loosely coupled and independent work units of different static sizes. Every computer with free capacity can request those work units -- in size or number corresponding to a user-defined capacity. The participator may complete the work unit offline or online. The completed work units are then delivered back to the server, where the results are verified. To make verification of results possible, the work units contain overlapping information and can be redistributed or recomputed. Successful and regular participators are rewarded with trust levels, so that they will be treated preferentially, i.e. that their results do not have to be checked rigorously and that they can obtain more work units.

IT Architect and developer view

This technology provides simple J2EE interfaces to developers to adapt applications in multi computer grids to distribute applications secure and reliable. Developers should be able to easily create grid-enabled parallel application without, themselves, becoming expert in grid or network computing. The operational model of the platform is a three-tier architecture: The web server, the application server, and the database are placed in three different zones. Zones are a representation of an area for which a common set of non-functional requirements has been defined, e.g. Firewalls. The rich client is implemented in Java and communicates only with the web server of the middle tier via stateless sessions using HTTP. In this concept, a client connects the server to synchronize the client version, to request work units, and to deliver completed work units. The server never connects a client to get any data. All interaction is initiated and defined by the participating user, since the user is the only person who can judge the free capacity of their resources. He can choose to run the client as a low-priority background process or in screensaver mode. ZetaGrid is optimized for all kinds and speeds of connections, e.g. modem, LAN, WAN.

Operational model

The following diagram outlines the operational aspects of the IT system and shows the key deployment units of the ZetaGrid platform and their placement on conceptual nodes. The nodes that are of interest in this document are highlighted in orange color and described in detail later in this document. Nodes of other platforms or external environments (marked in white) are briefly introduced in section ?. A description of the logical zones is contained in the next section. Actors are described briefly in section ?.



In the diagram above not only the nodes are of conceptual nature but the zones and even the deployment units as well. For zones it means that later they can be mapped to real physical zones. For nodes this means that, on a physical level, they can be separate units but do not have to. For instance, two or more conceptual nodes could be placed together on a shared physical node (machine). If so, the deployment units contained by these nodes cannot be distributed but have all to be placed on the same physical node. Vice versa, if multiple deployment units are placed on the same conceptual node, it is forbidden to deploy these units on different physical nodes. The placement of multiple deployment units on a single conceptual node is a design decision that enforces deployment of all deployment units on the same physical node.

Conceptual nodes

The conceptual nodes of the ZetaGrid platform are:

Conceptual Nodes of ZetaGrid	
Name of Node	**Description**
CN_ZetaGrid_Client	Rich client applications facilitating the IAP Client Container.
CN_WebServer	Hosts static content and routes requests for dynamic content to the following Web Application Server and back.
CN_WebAppServer	Hosts Web Applications based on Servlets, JSPs and pure Java Beans.
CN_DatabaseServer	Manages the repositories of the application data.

Each conceptual node and especially the contained deployment units are detailed later in this document.
The other conceptual nodes are not part of the ZetaGrid platform but are likely to be used to improve the performance and the availability of the ZetaGrid platform. The following table lists and briefly describes these other nodes:

Conceptual Nodes of related IAP Platforms	
Name of Node	**Description**

CN_Browser	Common web client that is usually a web browser.

Logical zones

The zones that are shown are just logical zones on a conceptual level and can be mapped to physical zones. A prerequisite for an assignment to real zones is the availability of specifications of concrete environments. The key characteristics of each of these zones are listed below:

Name	Description / Characteristic
User zone	Nodes in this zone are the accessing nodes to the application. They represent user workstation that might reside in the Intranet or Internet.
Demilitarized zone (DMZ)	Nodes in this zone must be protected by domain and protocol firewalls (on both sides) and might fulfill further security requirements like intrusion detection systems or SSL accelerators. This zone might also has nodes to improve performance requirements by reverse proxies caching static content to off load the http servers. Also, nodes in this zone might fulfill high availability requirements such that no single point of failure exists (using standby nodes and load balancers).
Web Application zone	Nodes in this zone host applications containing the business logic. Also, nodes in this zone might fulfill high availability requirements.
Database zone	The nodes in this zone are database servers that host the application data.
Administration zone	The nodes in this zone are to administer the other nodes. Since these nodes can interfere with productive operation, special considerations are required when defining the non-functional requirements of this zone.

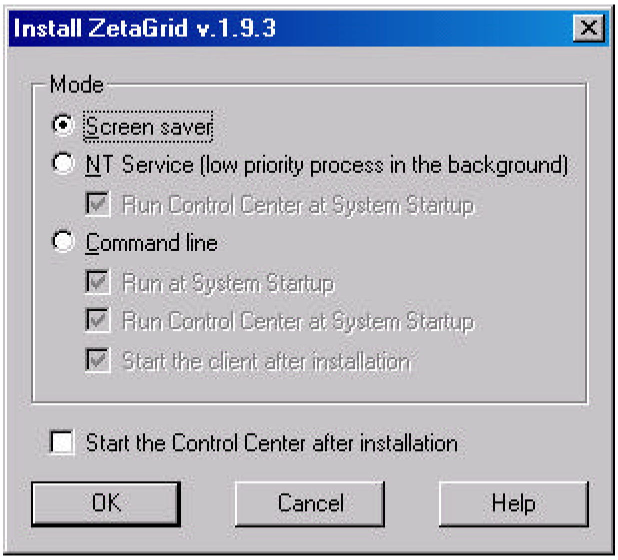
Roles

In ZetaGrid there are four different roles.

- A resource provider participates by using the ZetaGrid client.
- A developer builds a new task.
- A task administrator owns the task and can deploy and administrate the task.
- The system administrator is responsible for the global ZetaGrid server configuration. Only the system administrator has a special private key to activate new tasks in the system.

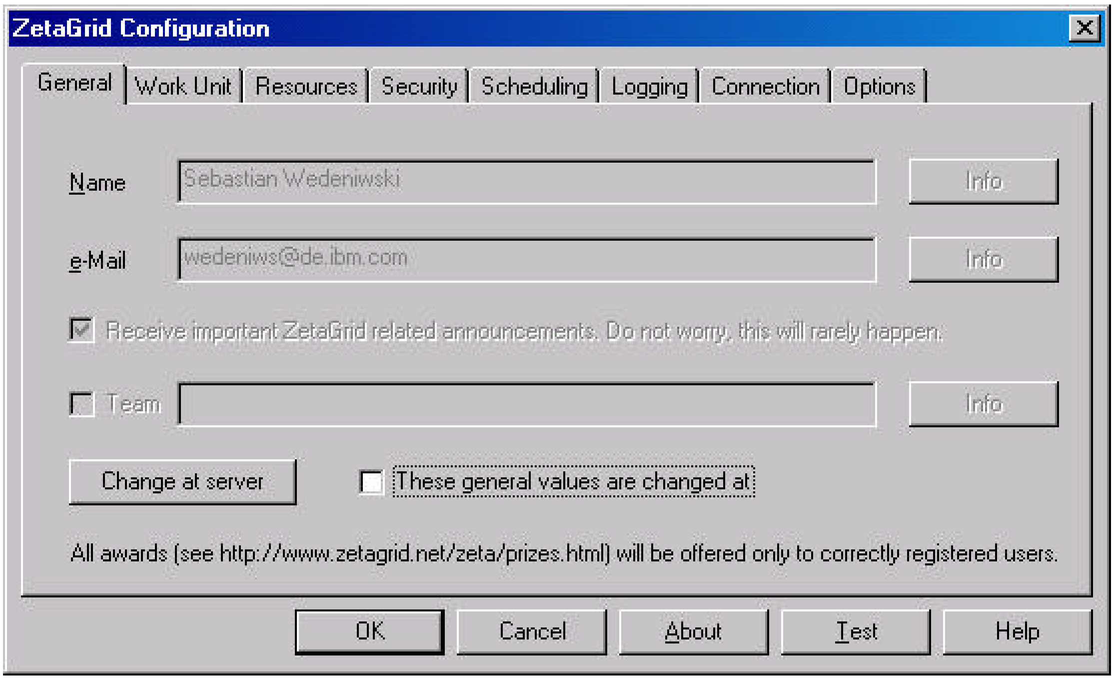
Configure the ZetaGrid client

The client can run in screensaver mode or as background process.
The client installation files are included in the ZetaGrid server module 'zetagrid.war'

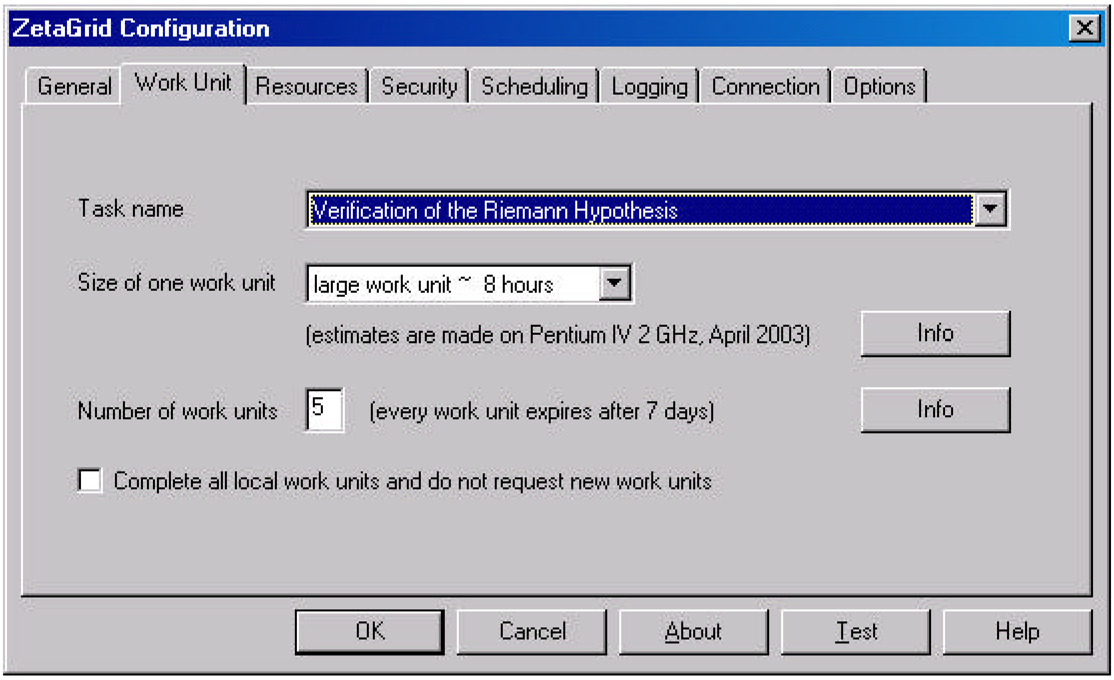


The client can be configured in the file 'zeta.cfg'

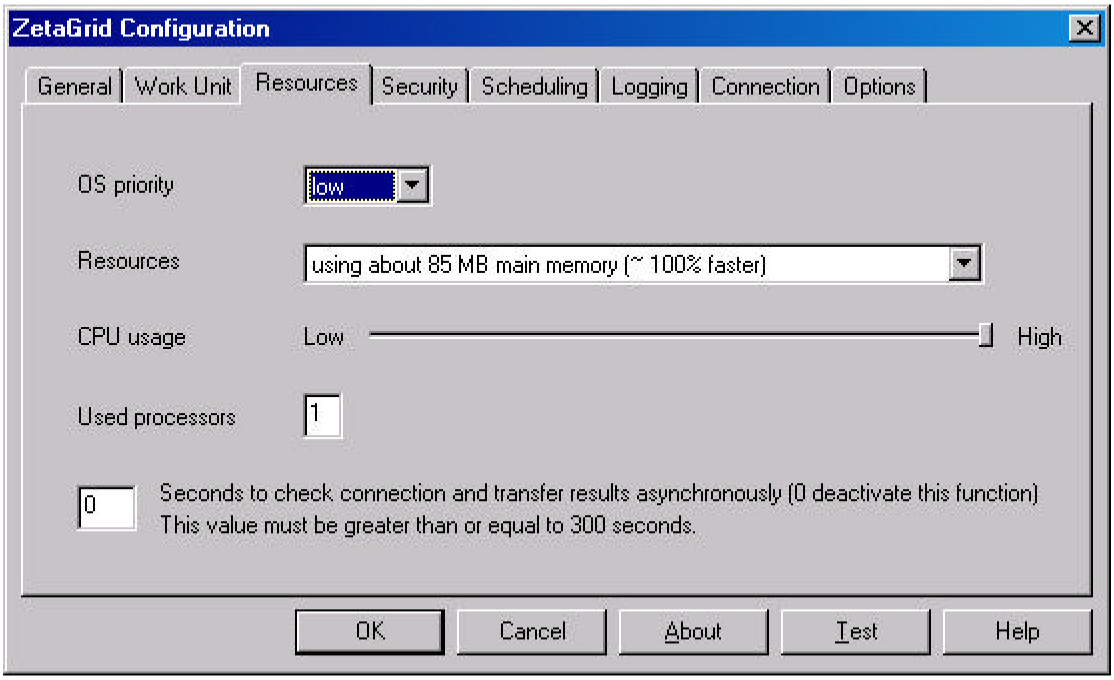
General



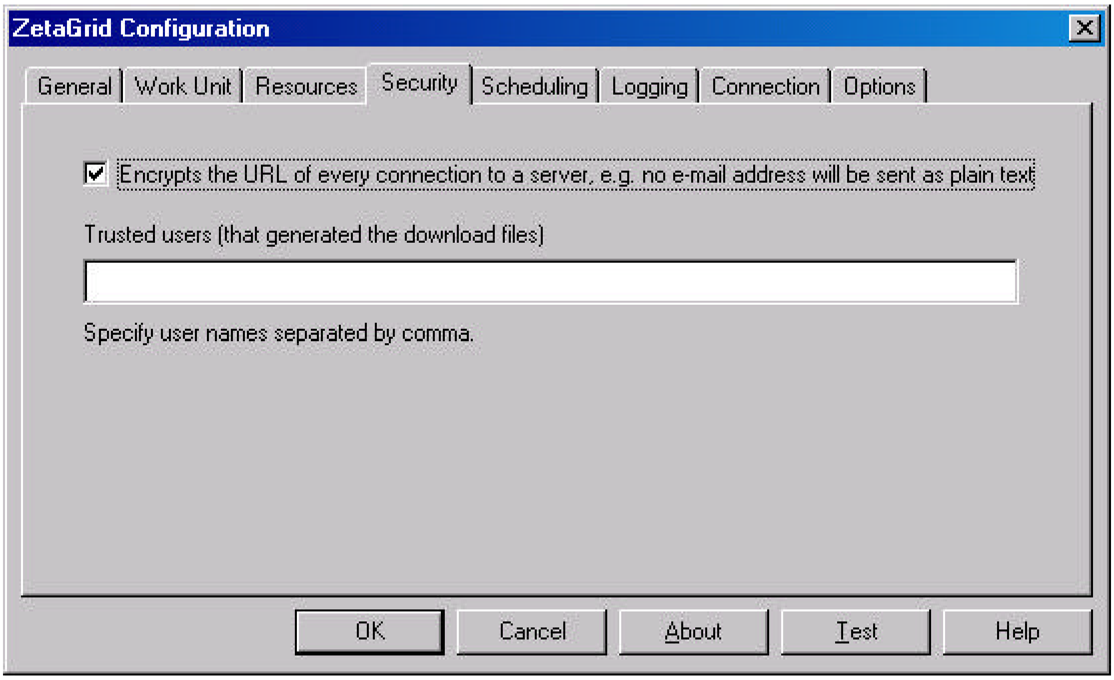
Work unit



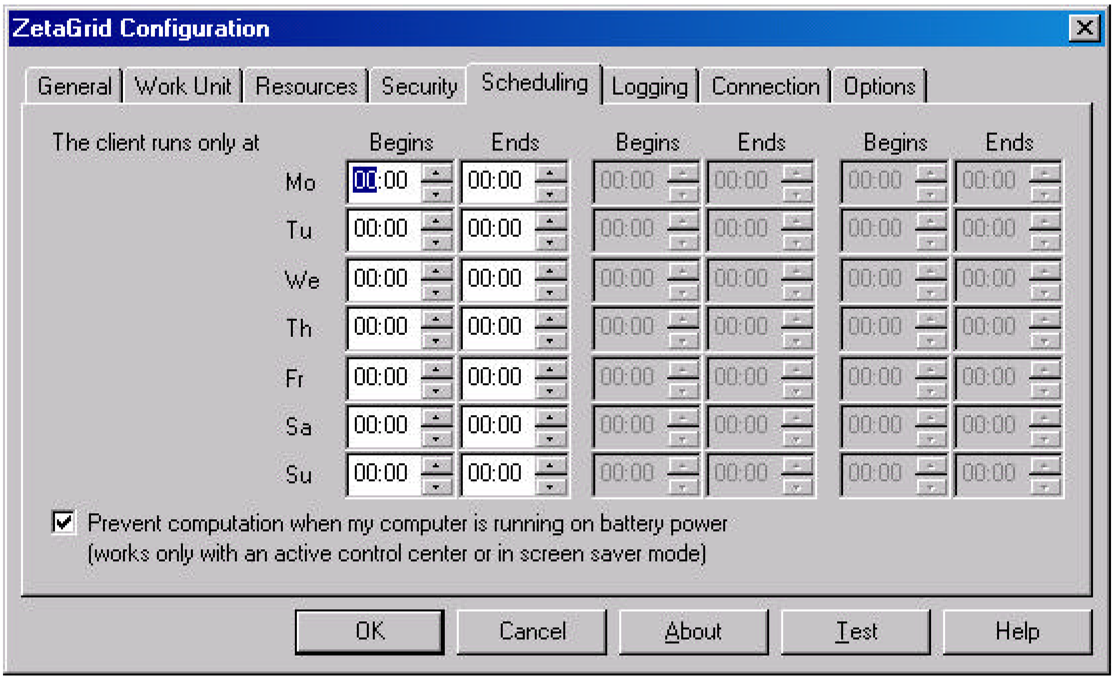
Resources



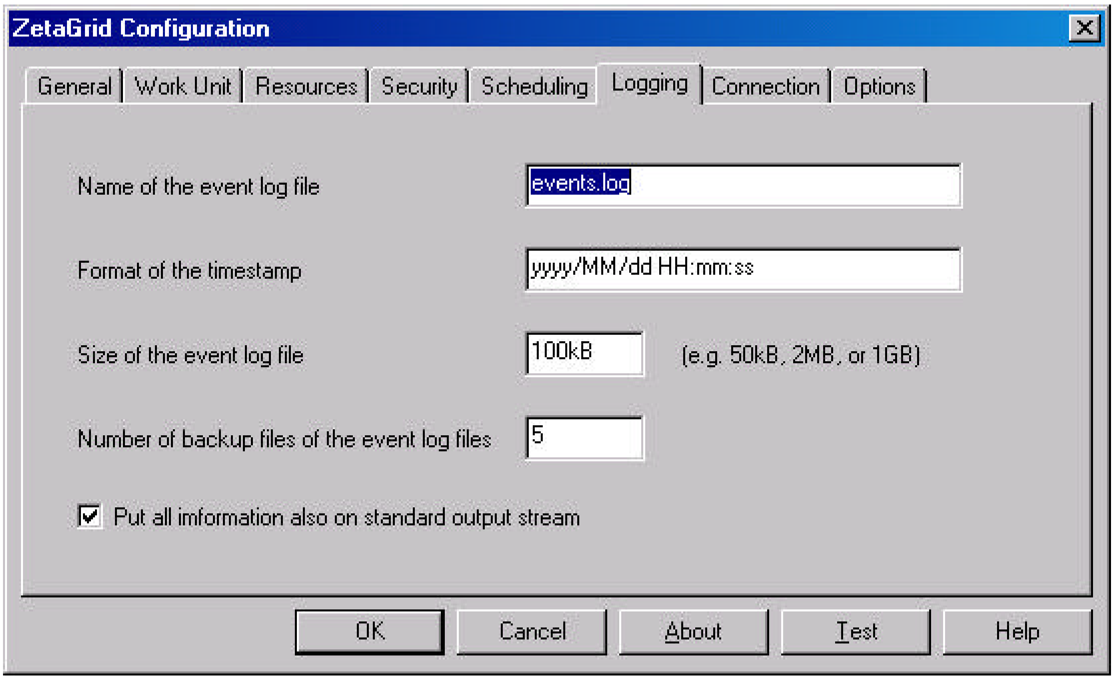
Security



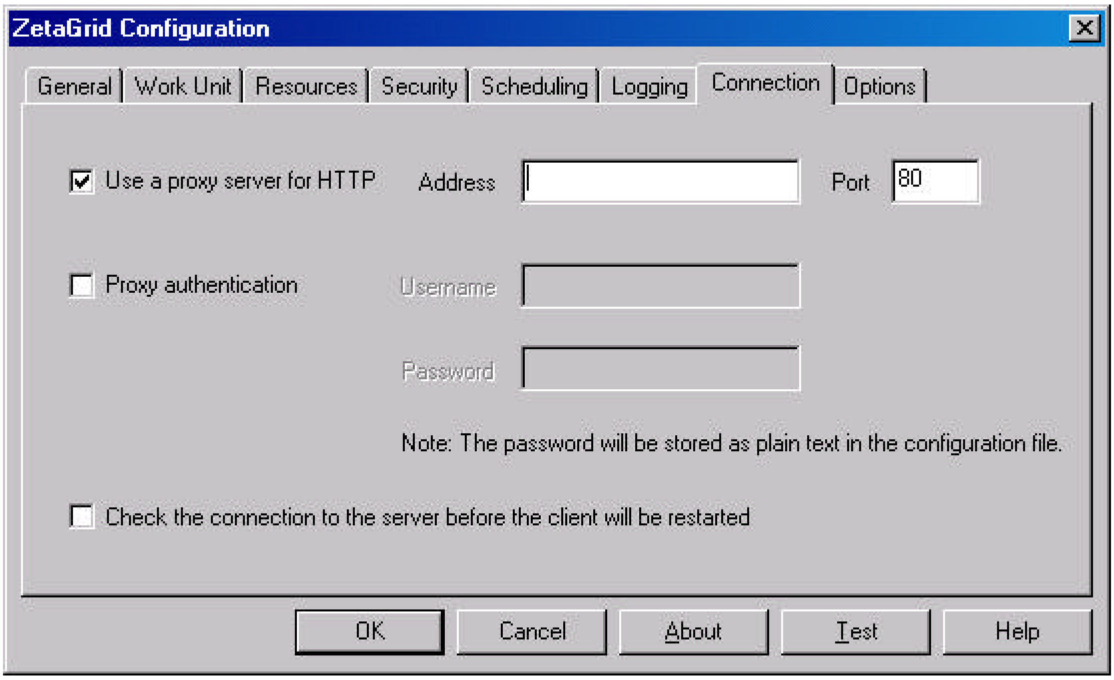
Scheduling



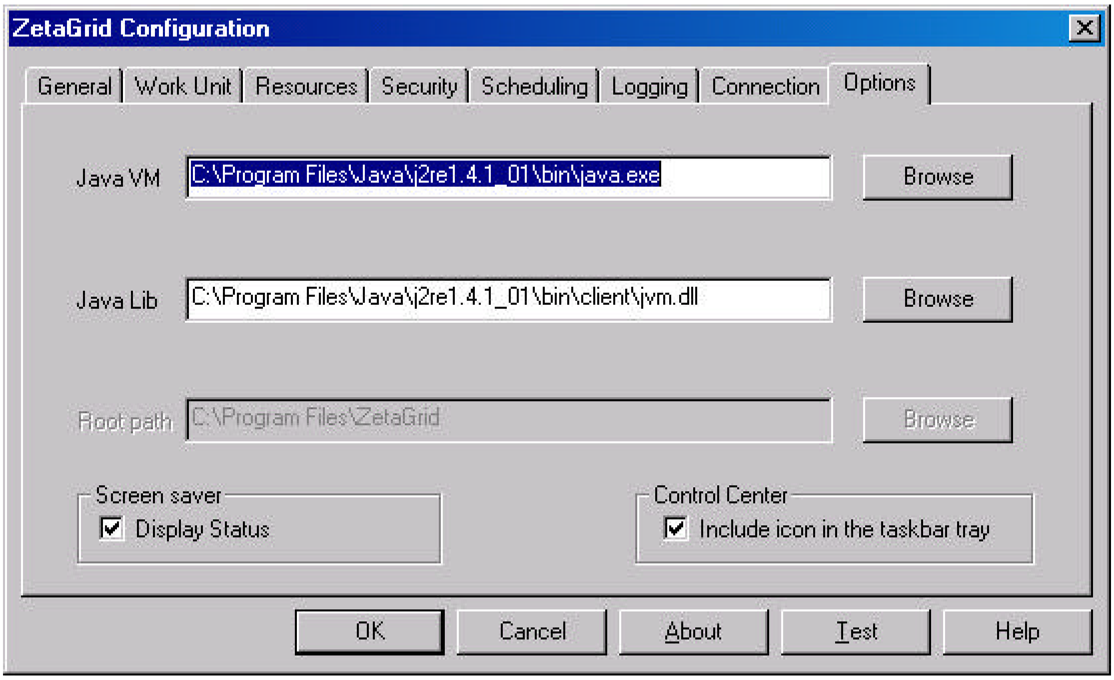
Logging



Connection



Options



How to develop a new task

A developer have to implement six classes:

1. A task that implements the interface 'zeta.ClientTask'
2. A work unit that implements the interface 'zeta.WorkUnit'
3. A task specific half-key to encrypt the results
4. Optional: A server-side work unit verifier that implements the interface 'zeta.server.WorkUnitVerifier'
5. Optional: A server-side request processor that implements the interface 'zeta.server.processor.TaskRequestWorkUnitProcessor'
6. Optional: A server-side result processor that implements the interface 'zeta.server.processor.TaskResultWorkUnitProcessor'

See the API documentation for further details.

The task interface

An example:

```
package zeta.example;

import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.List;

import zeta.Task;
import zeta.WorkUnit;
import zeta.ZetaInfo;
import zeta.util.Properties;
import zeta.util.StreamUtils;

public class ZetaTask extends Task {
  static {
    try {
      System.loadLibrary("zeta_zeros");
    } catch (Exception e) {
      ZetaInfo.handle(e);
    }
  }

  public ZetaTask(int id, String name, Class workUnitClass) {
    super(id, name, workUnitClass);
    try {
      properties = new Properties(Properties.ZETA_CFG,
Properties.DEFAULT_CFG);
    } catch (IOException ioe) {
      ZetaInfo.handle(ioe);
    }
  }

  public String getVersion() {
    return version;
  }

  public void setEnableStandardOutput(boolean enableStandardOutput) {
    setCoutLog(enableStandardOutput);
  }

  public void setResources(String resources) {
    if (resources != null) {
      try {
        setZetaResources(Integer.parseInt(resources));
      } catch (Exception e) {
        ZetaInfo.handle(e);
      }
    } else {
      setZetaResources(0);
    }
  }

  public int start(WorkUnit workUnit) {
    return zetaZeros(workUnit.getWorkUnitId(), workUnit.getSize(),
properties.get("sleep", 0));
  }

  public int stop() {
    try {
      zetaExit();
    } catch (Throwable t) {
      ZetaInfo.handle(t);
    }
    return 2000;
  }
```

```
  public List createWorkUnits(String parameters) {
    List workUnits = new ArrayList(10);
    int taskId = 0;
    long workUnitId = -1;
    int size = -1;
    boolean recompute = false;
    BufferedReader reader = null;
    try {
      reader = new BufferedReader(new StringReader(parameters));
      while (true) {
        String line = reader.readLine();
        if (line == null) {
          break;
        }
        if (line.trim().length() == 0) {
          taskId = 0;
          workUnitId = -1;
          size = -1;
          recompute = false;
        } else {
          if (line.startsWith("task_id")) {
            taskId = Integer.parseInt(line.substring(line.indexOf('=') +
1).trim());
          } else if (line.startsWith("work_unit_id")) {
            workUnitId = Long.parseLong(line.substring(line.indexOf('=') +
1).trim());
          } else if (line.startsWith("size")) {
            size = Integer.parseInt(line.substring(line.indexOf('=') +
1).trim());
            WorkUnit workUnit = createWorkUnit(workUnitId, size, recompute);
            if (workUnit.isValid()) {
              File file = new File(workUnit.getLogFilename());
              file.createNewFile();
              workUnits.add(workUnit);
            }
            taskId = 0;
            workUnitId = -1;
            size = -1;
            recompute = false;
          } else if (line.startsWith("recompute")) {
            recompute = true;
          }
        }
      }
    } catch (IOException ioe) {
      ZetaInfo.handle(ioe);
    } finally {
      StreamUtils.close(reader);
    }
    return workUnits;
  }

  public List createWorkUnits(String[] filenames) {
    List workUnits = new ArrayList(10);
    for (int i = 0; i < filenames.length; ++i) {
      if (filenames[i].startsWith("zeta_zeros_") &&
filenames[i].endsWith(".log")) {
        try {
          int idx = filenames[i].indexOf('_', 11);
          long workUnitId = Long.parseLong(filenames[i].substring(11, idx));
          int size = Integer.parseInt(filenames[i].substring(idx+1,
filenames[i].length()-4));
          WorkUnit workUnit = createWorkUnit(workUnitId, size, false);
          if (workUnit.isValid()) {
            workUnits.add(workUnit);
          }
        } catch (NumberFormatException e) {
        }
```

```
      }
    }
    return workUnits;
  }

  /**
   *  Contains a persistent set of the ZetaGrid properties.
  **/
  private Properties properties;

  private native static String getZetaVersion();
  private native static int zetaZeros(long workUnitId, int size, int
sleepN);
  private native static void zetaExit();
  private native static void setCoutLog(boolean coutLog);
  private native static void setZetaResources(int resourceId);
  private static String version = getZetaVersion();
}
```

The work unit interface

An example:

```
package zeta.example;

import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.ArrayList;
import java.util.List;

import zeta.WorkUnit;

public class ZetaWorkUnit extends WorkUnit {

  public ZetaWorkUnit(int taskId, long workUnitId, int size, boolean
recompute) {
    super(taskId, workUnitId, size, recompute);
  }

  public String getParameters() {
    StringBuffer buffer = new StringBuffer(8+9+13+19+6+9+7+9+2);
    buffer.append("task_id=");
    buffer.append(taskId);
    buffer.append("\nwork_unit_id=");
    buffer.append(workUnitId);
    if (recompute) {
      buffer.append("\nrecompute");
    }
    buffer.append("\nsize=");
    buffer.append(size);
    buffer.append("\nrange=");  // ToDo: remove
    buffer.append(size);
    buffer.append("\n\n");
    return buffer.toString();
  }

  public boolean isCompleted() {
    RandomAccessFile file = null;
    try {
      file = new RandomAccessFile(getLogFilename(), "r");
      long size = file.length();
      if (size > 0) {
        file.seek(size-1);
        if (file.readByte() == (byte)'@') {
          return true;
        }
      }
    } catch (IOException ioe) {
    } finally {
      if (file != null) {
        try {
          file.close();
        } catch (IOException ioe) {
        }
      }
    }
    return false;
  }

  public String[] containsFilenames() {
    return new String[] { "zeta_zeros_" + getWorkUnitId() + '_' + getSize()
+ ".txt", "zeta_zeros_" + getWorkUnitId() + '_' + getSize() + ".log" };
  }
  public String getLogFilename() {
    return "zeta_zeros_" + getWorkUnitId() + '_' + getSize() + ".log";
  }

  public boolean isPartOfWorkUnit(String filename) {
```

```
    return (filename.startsWith("zeta_zeros_") && (filename.endsWith(".log")
|| filename.endsWith(".txt")));
  }
}
```

The task specific half-key to encrypt the results

```
package zeta.crypto;

import java.math.BigInteger;

public class DefaultKeyEncrypt implements Key {
  public DefaultKeyEncrypt() {
    //public keys:
    p = new
BigInteger("12LBK5TMOB99C131JB3Q9UMVS18SFU7OOKL25K3OH51PL1H6GH3HDV6OIKL2LQI8
6KCB4LGMASDFLAPLJG5NJUV05ELC2PKLG2KG2SCB2", 32);
    p = p.shiftLeft(500);
    p = p.or(new
BigInteger("SGJLA1K936HBH0JVKKL4BQBL76G57KO1KH90345BILT4850I9ML5RG9Q8053KJAG
R7N8CJFLQI9LV0O15UV9INHUFMIII3LL5J53", 32));

    g = new
BigInteger("BLKT883FAB6V1BCF64G4PDUNMH9B9D317ECVHEAOK3OHPOS2E4D7CI8PK26EP36M
H1GJS1NG8B74D10G6E3KKJNI7QMFGR4Q8J7FQ8B8", 32);
    g = g.shiftLeft(500);
    g = g.or(new
BigInteger("C2NHRVQULGBJ8H4F2UU1GS0ATBJ5FDEIB7HGIN5U1K1ES0EB7NP8NQAPR41B5PQ1
TBOUS4PJ5AAC2GRSJSEMKBEN49IGHOA8ORFA", 32));

    A = new
BigInteger("U53EO946TT0EQ0QHGS6GCBBUVVDRF9U50QVB3795MIP3OH18S91AAFQV54N5RFBL
3I7UIPAUM4R67PUQEV3JQH9PNPEJ84IU6L3FVM7L", 32);
    A = A.shiftLeft(500);
    A = A.or(new
BigInteger("9VU1F4R6TDV9HQH0RAKL7VQEI3G87M4F86CRQKQ9P5C892F2QPU1M6UCSEIDL2VP
8UDK9808UHHGVVPVFBVD6AE7NC8DJHSUKIEG", 32));
  }

  public BigInteger getBase() {
    return A;
  }

  public BigInteger getModulo() {
    return p;
  }

  public BigInteger getGenerator() {
    return g;
  }

  private BigInteger p,g,A;
}
```

A server-side work unit request and result processor

```
package zeta.server.processor;

import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.IOException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.SQLException;
import java.util.Map;
import java.util.zip.ZipInputStream;
```

```
import javax.servlet.ServletException;

import zeta.WorkUnit;
import zeta.util.StreamUtils;

/**
 *  The default processor for work units which are received through the
request and the result handler.
**/
public class DefaultWorkUnitProcessor implements
TaskRequestWorkUnitProcessor, TaskResultProcessor {
  /**
   *  Checks work units received through the result handler
   *  @param workUnit work unit which should be checked
   *  @param result buffer with the zipped result
  **/
  public void checkResult(WorkUnit workUnit, byte[] result) throws Exception
{
    ZipInputStream zip = null;
    try {
      zip = new ZipInputStream(new ByteArrayInputStream(result));
      if (zip.getNextEntry() == null) {
        throw new IOException("empty");
      } else {
        while (zip.getNextEntry() != null);
      }
    } finally {
      StreamUtils.close(zip);
    }
  }

  /**
   *  Processes work units received through the result handler
   *  @param stmt statement object's database
   *  @param workUnit work unit which should be processed
   *  @param result buffer with the zipped result
   *  @return <code>true</code> if the ResultHandler shall save the result
into the database.
   *  @exception  IOException  if an I/O error occurs.
  **/
  public boolean processResult(Statement stmt, WorkUnit workUnit, byte[]
result) throws ServletException, SQLException, IOException {
    return true;
  }

  /**
   *  Returns the parameters which are associated with the specified work
unit; are separated by the character ','
   *  @param workUnit work unit
   *  @return parameters which are associated with the specified work unit;
are separated by the character ','
  **/
  public String getParameters(WorkUnit workUnit) {
    return null;
  }

  /**
   *  Activates the specified work unit for the requested client.
   *  @param stmt statement object's database
   *  @param workUnit work unit
   *  @return less than 0 if an error occurs, 0 if the specified work unit
is activated but no further work unit can be activated,
   *          and greater 0 if the specified work unit is activated and
further work units can be activated.
   *  @exception  SQLException  if a database access error occurs.
  **/
```

```
  public int activateWorkUnit(Statement stmt, WorkUnit workUnit) throws
ServletException, SQLException {
    return 1;
  }
}
```

How to set up a ZetaGrid server

The ZetaGrid server requires a database server (e.g. IBM UDB 8.x) and a Web Application server (e.g. Tomcat 4.x.).

Specify two database users “zeta” and “zetacalc”
For example you can specify <name of the zeta database> = “zeta”

1. Create the database <*name of the zeta database*> by the command
   `db2 create db <name of the zeta database>`
2. Connect to the new database <*name of the zeta database*> by the command
   `db2 connect to <name of the zeta database>`
3. Setup the tables by the command
   `db2 –tf  zeta_db.sql`
4. Adjust the deployment descriptor ‘web.xml’ (in the folder ‘WEB-INF’) of the Web Application ‘zetagrid.war’ (e.g. specify user ID and password).
5. Install the adjusted file ‘zetagrid.war’ at your Application Server.
6. Configure your servers in the table ‘zeta.server’
   it is easier to configure loosely coupled servers with no synchronization.

How to deploy a new task

Define a task in the table ‘zeta.task.’ The following values must be defined:

```
--=====================================================================
CREATE TABLE zeta.task
--=====================================================================
(
  id                        INTEGER NOT NULL,
--.....................................................................

  name                      VARCHAR(100) NOT NULL UNIQUE,
  client_task_class_name    VARCHAR(250) NOT NULL,
  work_unit_class_name      VARCHAR(250) NOT NULL,
  encryption_class          LONG VARCHAR NOT NULL,
  encryption_signature      VARCHAR(1000) NOT NULL,
  decryption_number         VARCHAR(1000) NOT NULL,
  request_processor         VARCHAR(250) NOT NULL,
  result_processor          VARCHAR(250) NOT NULL,
  verifier_class_name       VARCHAR(250),

  PRIMARY KEY(id)
);
```

Every task may contains various programs which can be defined in the table ‘zeta.program.’ This table should only be changed by the utility ‘zeta.server.tool.NewVersion’ to insert, update, or remove a program from a task. This utility needs a global private key which should only be known by the system administrator.

```
--=====================================================================
CREATE TABLE zeta.program
--=====================================================================
(
  task_id                 INTEGER NOT NULL,
  name                    VARCHAR(100) NOT NULL,
  os_name                 VARCHAR(100) NOT NULL,
  os_version              VARCHAR(20) NOT NULL,
  os_arch                 VARCHAR(20) NOT NULL,
  processors              SMALLINT NOT NULL,
--.....................................................................

  version                 CHAR(4) NOT NULL,
  key_class_name          VARCHAR(250) NOT NULL DEFAULT
'zeta.crypto.DefaultKey',
  program_from_user       VARCHAR(100) NOT NULL,
  compressed_YN           CHAR(1) NOT NULL,
  program                 BLOB(5M),
  last_update             TIMESTAMP NOT NULL DEFAULT CURRENT TIMESTAMP,
  signature               VARCHAR(1000) NOT NULL,

  PRIMARY KEY(task_id, name, os_name, os_version, os_arch, processors)
);
```

Furthermore, the specified client work unit size can be mapped on a real size in the table ‘zeta.work_unit_size.’

```
--=====================================================================
CREATE TABLE zeta.work_unit_size
--=====================================================================
--
-- Defines the range of a work unit for a specified task and size with
-- the id less than or equal to 'work_unit_id'.
--
-- For the task 'zeta-zeros' we have the sizes:
-- t: tiny work unit ~  60 minutes
-- s: small work unit ~  90 minutes
-- m: medium work unit ~ 3 hours
-- l: large work unit ~  4 hours
-- h: huge work unit ~  6 hours
--
(
  task_id                 INTEGER NOT NULL,
  size                    CHAR(1) NOT NULL,
  work_unit_id            BIGINT NOT NULL,
--.....................................................................

  range                   INTEGER NOT NULL,

  PRIMARY KEY(task_id, size, work_unit_id)
);
```