



## xTier™ Grid Service

The xTier™ Grid Service provides a simple yet robust implementation for developing Near Real-time (nRT) Computational Enterprise Grids.

### Key Benefits:

- Simple yet powerful implementation
- Cross-paradigm grids
- Aggregate a variety of computing resources into a single unified pool of processing power
- Highly cost-effective for large *and* mid-size grid implementations
- Develop functionality that couldn't be implemented in the enterprise due to processing time constraints

## Introduction

Every task solved places varying demands on the hardware upon which it is executed. Typically when a given task exceeds the processing power, memory, or network bandwidth of the underlying hardware new more powerful hardware must be purchased. There exist, however, many problems which exhaust all of the processing power available on a single computer. The solution to this problem is to change the task such that it can be solved by several computers working on the task in parallel.

Distributed computing has many different definitions, but simply stated it is a method that spreads a complex computational task over several computers. Depending on the task type and the software being used, the computers used for this computation can be located in a single office or scattered around the globe.

Recently the term "Grid Computing" has gained prominence in the marketplace. There is a wide variety of different technologies and as many definitions as to what Grid Computing should be. Generally, however, Grid Computing can be defined as: "a form of distributed computing that involves coordinating and securely sharing computing, storage, and network resources."

As Grid Computing means a variety of things to many different people, it is useful to examine the various technologies through two different classification schemes; 1) by considering the intended purpose of the grid product and 2) by considering the intended audience of the grid product. In the section that follows, the various grid products and paradigms are discussed with respect to these two classifications.

## Grid Systems Classified by Purpose

First, consider the purpose of the grid system in this way there are two main types of grid: ***Computational Grids***, and ***Data Grids***.

A **Computational Grid** is a middleware product that allows using computational resource distributions. The main computer which receives the computational task for execution (the Grid Server) distributes it among the Grid Workers (the computers connected in the grid system). The Grid Workers receive their task portions, compute the answer to the sub-task and return the result back to the Grid Server. Special software located on the Grid Server provides the facilities to split the tasks into smaller sub-tasks and then distributes these sub-tasks to the Grid Workers. The Grid Server accepts the sub-task results and then aggregates all of these results into the final result.

**Data Grids** are middleware that tie together storage systems that are distributed across administration domains and are linked by Wide Area Networks (WANs). The term "Data Grid" traditionally represent the network do distributed storage resources, from archival systems, to caches, to databases, that are linked using a logical name space to create global, persistent identifiers and provide uniform access mechanisms. A Data Grid's user can store huge amounts of information without consideration for the physical data allocation on particular computers and can also perform complex data searches in the whole data grid system without consideration for individual repositories. It is important to note that Data Grids should not be confused with a simple system of multiple hard drives sharing in a local network. Data Grids consider the wider implications of data and storage aggregation along with distribution, management and fast data searching.

While with computational grid user deals with computational task distribution, with data grids deals with information (data) distribution. When examined in light of this classification, the xTier™ Grid Service is classified as a Computational Grid.

## Grid Computing and "On-demand Computing"

Grid computing and on-demand computing should not be confused. On demand computing is concerned with virtualizing a large number of computing resources into a pool of common system resources. As such a given user can request some systems resources and these resources will be allocated dynamically from this pool for the User.

The user will not know what computing resources or disk storage they are using to service this request as all of the computing resources in a given on demand architecture are aggregated into a "virtual" computer. The user's tasks are executed by this virtual computer and the user does not work with a physical hard disk, but rather with common virtualized data storage.

Resource virtualization improves manageability – the architecture that the end user utilized moves from servers, ports and discs to virtual volumes, virtual computational resources and virtual networks. As long as these virtual resources are available, the underlying hardware can be added to, removed or reset without interruption of the end user experience. The user does not care where a given task is executed, merely that it is executed; the user just consumes a certain amount of virtual processor power from the on demand infrastructure.

## Grid Systems Classified by Audience

The second classification that is useful to further differentiate the different types of grids is the intended audience to which a given grid implementation is directed. In this case there are two different audiences: **Enterprise Grids** and **Global Grids**. An **Enterprise Grid** is a grid in which corporate level computer networks combining one or several groups of machine are working on the same or different projects. A **Global Grid** is a grid which networks several independent organizations together to pool and share resources with each other. The participating organizations set the rules of resources exchange and certain interaction protocols.

### Global Grid Review

Global Grid systems are created to solve long-running, complex computationally intensive tasks. Generally these tasks are associated with academic research projects or are connected with some scientific activity. Grids of this type connect large numbers of computers, often numbering in the thousands, from around the globe and the calculations can last for years.

The following are some common examples of this type of grid:

- SETI@home - <http://setiathome.berkeley.edu/> , extraterrestrial intelligence search.
- Cancer Research Project - <http://www.chem.ox.ac.uk/curecancer.html>, search for the cure for cancer.
- Climate Prediction - <http://www.climateprediction.net>, weather forecast for next 59 years.
- The Globus Alliance – [www.globus.org](http://www.globus.org), toolkit for developers of global grid applications.

These projects use global resources to solve problems from medical, mathematical, financial and various other scientific areas. All computers taking part in projects such as these have client software installed on each computer. Practically any computer, even with different architectures, can become a client of these networks as long as there is a client application for the given configuration. The most well-known public example of grid computing, SETI@Home, implemented the client as a screen saver for the Windows operating system and consequently could utilize all of the free processor time on all of the client computers.

### Enterprise Grid Review

Enterprise Grid systems are created in order to solve business applications within a given enterprise. Typically these tasks do not have a long-running nature (as compared to the global grid tasks) but rather are used when execution time of a given task is beyond acceptable levels using standard tools and hardware. Additionally, by implementing an Enterprise Grid it is possible to do computations that were previously unavailable to the enterprise as they could not be solved using the existing hardware and infrastructure.

Enterprise Grids have begun to receive attention in the marketplace with many corporations are interested in utilizing the wasted processing power that is currently underutilized within their datacenters. Enterprise Grids are typically utilized to perform functions which are not possible using application deployed with conventional architecture or when the scalability limits of these applications deployed on traditional architecture are met. For instance, if a company wants to provide some information on its web-site upon user request but processing this request takes too much time usually company will

just drop such functionality. In interactive applications users don't want to wait and response times are very important for usability. Choosing between richness of functionality and usability companies often prefer the last and they just skip some functions reducing their web sites useful features. Using grid technologies it is possible to reduce information processing times tenfold and more thus allowing companies to provide broader functionality.

### **Near Real-time Grid Applications**

Most of the traditional grid systems were designed to solve extremely computationally expansive problems that could take months if not years to finish. These types of problems appear predominately in scientific research and a few specific business areas such as Biotech. The majority of business grid applications, however, would have very different scale of computational duration, for instance tasks which usually take 10-15 minutes and need to be reduced to 5-10 seconds. These types of tasks are Near Real-time (nRT) level tasks. For this type of task it is very important that the processes of task splitting and results aggregation are as fast as possible. For long running scientific tasks (taking months and more) it is not as important as to how much time will be spent splitting, distributing to execution nodes and aggregating as it is to perform the calculations as quickly as possible because the calculations themselves take an order of magnitude more time than these auxiliary activities. On the other hand, it is critical to reduce these times as much as possible as nRT Grid applications are sensitive to these activities as a result of their comparatively smaller executing times and processing throughput should be provided for solving the task in a high-performance manner and not spent on auxiliary activities.

### **xTier™ Grid Service**

The current solutions in the marketplace impose significant architectural structures on the corporate IT environment. Further, they are costly to purchase and implement. Currently, trying to use existing grid solutions for relatively small business tasks can lead to a big project requiring significant human resources for implementation of the Grid infrastructure and integration and implementation with the software interfaces.

The need to supply a simple to use, powerful, cost effective grid infrastructure is apparent as the majority of existing solutions are too costly, or too complex for small to mid-size companies to use, or for small to medium grid implementation. Further, for enterprise grid computing, the toolset itself should be a unified, simple and extensible implementation that easily integrates with existing code bases so that implementing a grid for is as cost and time effective as possible. The xTier™ Grid Service takes all of these factors into account and provides a simple and very effective service for enterprise grid computing.

Furthermore, one of the unique characteristics of the xTier™ Grid Service is its support for near real-time (nRT) grid applications. The xTier™ Grid Service is the only grid infrastructure that was designed from the ground up to address both types of computational intensive problems - traditional long running problems and nRT grid applications.

The major characteristics of the xTier™ Grid Service (an enterprise computing grid infrastructure) are:

- Direct support for task splitting and aggregation
- Easy to use, configure and manage
- Allows developers to customize and extend the supplied implementation to tailor the xTier™ Grid Service for very specific environments.
- Support for near real-time applications
- Support for dynamic allocation of computing resources (using the supplied xTier™ Cluster Service)

- Seamless integration with the entire xTier™ Service collection
- Cross-paradigm implementation for mixed run-time grid deployments.

### xTier Grid service - Basic Terms and Definitions

xTier Grid Service is an enterprise computing grid solution as was classified above. To understand how the xTier™ Grid service works there are two basic concepts to understand.

- *grid task* – the task that is registered in the service and that will be distributed for execution.
- *task unit* – the task part distributable to the grid cluster nodes for execution.

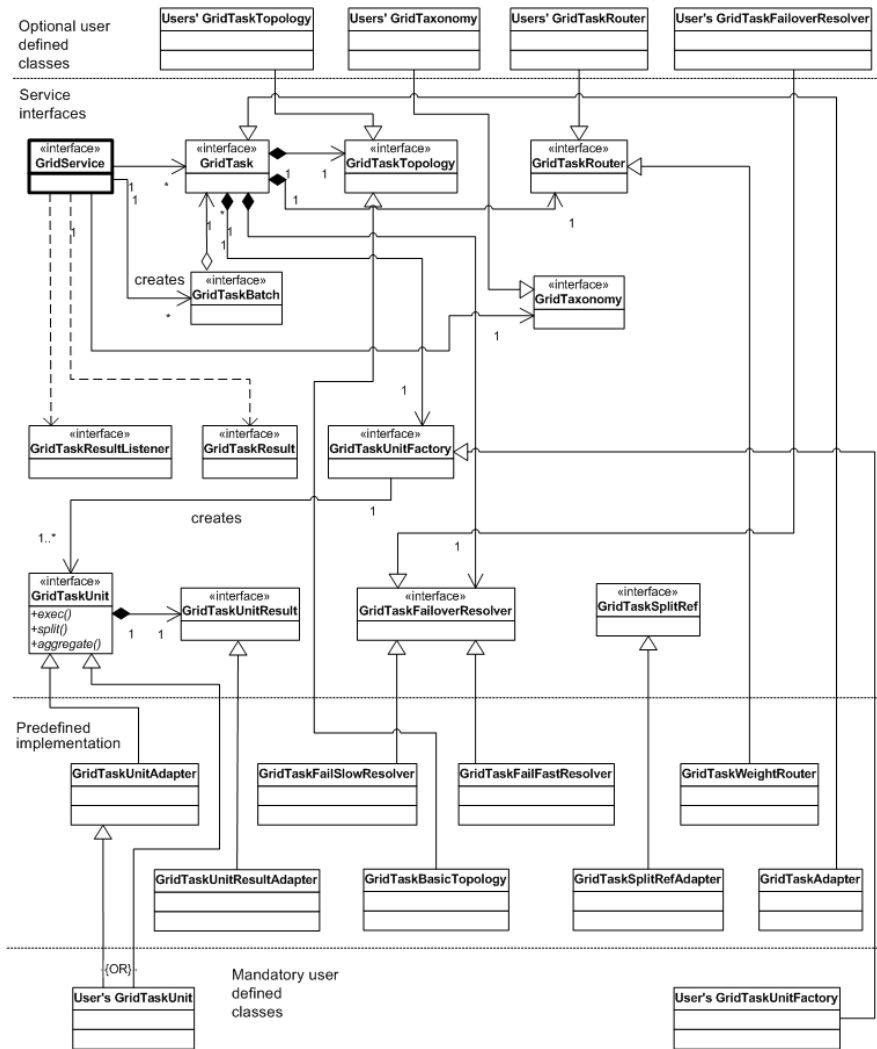
A root task representing the entire task is created. It can then be split into sub-tasks with each of the sub-tasks splittable as well. As a result the whole task is decomposed into a tree-like structure with every node of representing a task unity where each task unit represents a piece of work that can be distributed over the network for execution on the other grid nodes. Note, the developer defines the logic of splitting the task unit into sub-units (implementation details will be described below).

Each grid task is identified by its ID. Note that this ID should be the same on all grid nodes and it should "survive" xTier™ restarts. In most cases this ID is directly coded in the source code or in XML configuration. Tasks get registered, de-registered and executed via the **GridService** interface, which is the service's base interface.

A grid task consists of the following sub-components that fully define how this task will be executed:

- **GridTaskTopology** (topology resolver) - This component provides the facility to obtain a list of grid nodes for a specific task unit effectively defining the sub-grid on which a given task unit must be executed. The grid topology resolver is very important for efficient grid execution because at any point of time it should produce the sub-grid for a specific task unit that will provide the most effective task load distribution. In many cases, the implementation of the topology resolver would need to actively monitor the grid cluster and collect certain statistics about task execution to determine the best distribution on a given time.
- **GridTaskRouter** - This component works together with the grid topology resolver. When the topology resolver returns a set of grid nodes for a specific task unit, the task router's responsibility is to determine the specific node where the task unit will be sent for execution. In many cases, the task topology and task router should work in concert, sharing certain activities yet separating their responsibilities.
- **GridTaskUnitFactory** - This component is responsible for creating task units for this grid task.
- **GridTaskFailoverResolver** – This component determines the failover logic for the task units within this grid task.

The following diagram provides and overview of the classes that make up the xTier™ Grid Service.



### Optional Service Elements

The following are optional service elements utilized with the xTier™ Grid Service.

Service interfaces:

- GridTaskBatch** - combines one or more task grids for batch execution. Note that batch execution is always sequential and grid tasks will be executed in the same order they were added to this batch. GridTaskBatch provides simple execution grouping for grid tasks.

- **GridTaxonomy** - provides the relative characteristics of the nodes in the grid to achieve more optimal sub-task (task units) distribution on a given grid topology.
- **GridTaskResult** - result of the grid task execution.
- **GridTaskResultListener** - defines the callback for asynchronous grid task execution.
- **GridTaskUnitResult** - defines the task unit execution result.
- **GridTaskSplitRef** - defines the task unit references. This reference is used to route the referenced task unit to a remote cluster node for execution. Split references are a fundamental part of the grid service operation: grid nodes exchange split references instead of exchanging the task unit themselves. Each split reference contains information about the task unit it references, the argument for the task unit execution, optional routing information, as well as relative weights that show how this split reference relates to other split references in terms of CPU, IO and memory load.

#### Predefined Implementations (Supplied with xTier):

- **GridTaskUnitAdapter** - defines a convenient adapter for a grid task unit which simplifies custom implementations.
- **GridTaskUnitResultAdapter** - defines a convenient adapter for a grid task unit result.
- **GridTaskFailSlowResolver** - defines a fail-slow grid task failover policy. The following table describes the mapping between a task unit result error code and the failover mode applied.
- **GridTaskFailFastResolver** - defines a fail-fast grid task failover policy.
- **GridTaskBasicTopology** - defines a basic topology resolver. This topology resolver subscribes for cluster event to receive notifications when a new node is added to the cluster or an existing node leaves the cluster. It also allows the topology to be filtered based on cluster group participation as well as the CPU utilization threshold from the remote nodes. It also allows the topology to be filtered by local node, local host nodes and remote nodes.
- **GridTaskSplitRefAdapter** - defines a convenient adapter for a grid task split reference.
- **GridTaskWeightRouter** - defines a weight-based grid task router. This grid task router routes split references based on the best match between weights specified in the split references and the nodes' taxonomy available from the grid service.
- **GridTaskAdapter** - defines a convenient adapter for a grid task.

Interfaces that must be implemented by user:

- **GridTaskUnit**
- **GridTaskUnitFactory**

For the majority of tasks involving the grid service it will be enough to implement these 2 interfaces only.

#### Optional elements that can be implemented by user:

Custom **GridTaskTopology** – if the developer needs custom logic different than implemented in the predefined implementation of **GridTaskBasicTopology**, they can create a custom implementation of the **GridTaskTopology** interface to fit the specific scenario.

Custom **GridTaskRouter** – if the developer needs custom logic different than implemented in the predefined implementation **GridTaskWeightRouter**, they can create a custom implementation of the **GridTaskRouter** interface to fit the specific scenario.

Custom **GridTaskFailoverResolver** – if the developer needs custom logic different than implemented in the predefined implementation **GridTaskFailFastResolver** and **GridTaskFailSlowResolver**, they can create a custom implementation of the **GridTaskFailoverResolver** interface to fit the specific scenario.

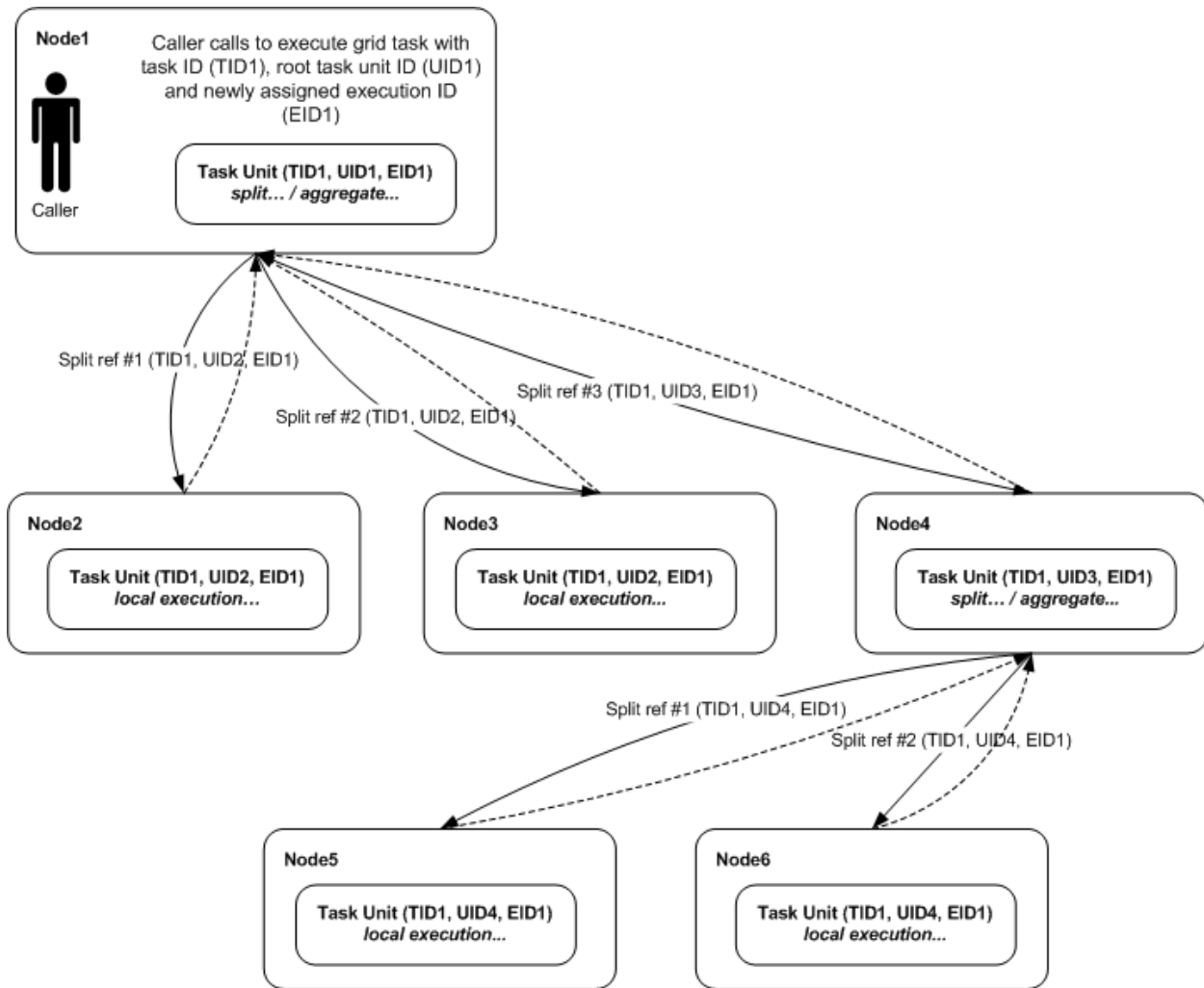
Maximum system flexibility is achieved in this way as the developer has the option to tailor the grid environment to their specific implementation and architectural needs.

### Grid Task Execution Process

To better understand the semantic of how grid execution works with the xTier™ Grid Service, let's examine an example of how a grid task gets executed. (Note the following example does not depict the actual algorithm but rather presents a simplified user view on how a grid task logically proceeds with execution in the service.)

This process can be illustrated by the following steps and diagram:

- 1) When using the xTier™ Grid Service the developer calls the **exec(GridTask, Marshallable)** method from the main service interface. The **GridTask** parameter represents the execution task, the **Marshallable** parameter represents the cross-platform container for the execution arguments.
- 2) Each execution of a specific grid task is assigned a new Execution ID (EID). EIDs are unique across the grid cluster. A new EID is created and assigned to this execution.



- Each grid task has an associated tree-like structure of task units. Each node in this tree is a task unit, and a task unit represents a unit of work that can be distributed over the network to another grid node for execution. The root task unit is obtained by calling the **GridTaskUnitFactory.newTaskUnit(int, int, long)** and passing the **GridTaskUnit.ROOT\_TASK\_UNIT\_ID** constant for task unit ID (UID).

At this point in the execution cycle, a Grid task, New EID, and Root task unit have been obtained. From this point onward the work of 'grid' service is to basically take a root task unit, determine if it needs to be split and process its sub units, if any. This process repeats itself recursively on all involved grid nodes until the final result is fully aggregated back to original grid node and returned.

- With the grid task ID, EID and root UID we can obtain the topology for the specific (root) task unit by calling the **GridTaskTopology.getNodes(int, int, long)** method on the topology resolver from the grid task. If the topology returned is empty, then the error task unit result is returned with the error code

**GridTaskUnitResult.ERR\_RETRY\_OTHER.** If the topology returned has only one node and this node is the local node, then the current task unit is forced to execute locally by calling the **GridTaskUnit.exec(Marshallable)** method and the execution result is returned.

- 5) At this step, we have a grid task, (root) task unit ID and EID with a non-empty topology. Since we have a non-empty topology the grid service assumes that this indicates that there are available on-demand resources that can be used for asynchronous execution of this task unit. The method **GridTaskUnit.split(Set, GridTaxonomy, Marshallable)** is called to perform a split of the given task unit into a set of sub task units. This method takes a non-empty topology, grid taxonomy and the arguments that were passed in with the task unit and produces either null or a set of **GridTaskSplitRef** instances. If it produces null it indicates that the task unit rejected the split and it should be executed locally; such task unit are executed locally by calling the **GridTaskUnit.exec(Marshallable)** method and its result is returned.
- 6) At this point in the execution cycle we have obtained a set of **GridTaskSplitRef** instances. Each split reference references a sub task unit. It is important to note that **GridTaskSplitRef** instances only reference a given task unit but do not define it. In fact, it does not have splitting or execution logic in it. This distinction is very important because the grid service uses these references to distribute each task unit execution onto the remote grid nodes: instead of sending the whole task unit it sends only the reference to it and on the receiving node the referenced task unit will be recreated (using **GridTaskUnitFactory**) and further split or locally executed with similar steps as described above. The way by which every split reference is assigned to a specific node is determined by the **GridTaskRouter** instance available from the grid task. Notice that the task router and split reference have an API to work together: the split reference can have optional router information (see the **GridTaskSplitRef.getRouterInfo()** method) that can be used by a specific router implementation. In that way a task unit can pass certain information during its split to the router to affect the routing of a specific task unit reference.
- 7) Once the split reference is obtained along with its destination grid node, the grid service sends this reference to that node for execution. The execution steps on the remote node are identical to the ones above. In the end, the remote task unit execution can have three outcomes:
  - Execution is timed out. In this case the original node will try to failover using the task's failover resolver.
  - An error result is returned. In this case the original node will try to failover using the task's failover resolver.
  - A positive result is returned. In this case, when all results are returned positively the original node will aggregate these results using the original task unit's **GridTaskUnit.aggregate(Set)** method. The aggregated result will then be returned.
- 8) When root task unit is successfully executed, its task unit result is converted to task result and returned back as grid task execution result.

## Service Configuration

The xTier™ Grid Service is configured via the XML configuration file **xtier\_grid.xml**. This file follows standard configuration template of xTier service. Here is a complete example of xTier™ Grid Service configuration:

```
<xtier-grid>
  <region name="example">
    <config>
      <ip-port default="54321"/>

      <thread-pool-name>grid.thread.pool</thread-pool-name>

      <!-- Specifies how many execution traces a grid node keeps. -->
      <max-exec-traces>100</max-exec-traces>

      <!-- Optional grid taxonomy. -->
      <!--
      <taxonomy>
        <ioc uid=" my.taxonomy" policy="new">
          ...
        </ioc>
      </taxonomy>
      -->
    </config>

    <!--
    Grid task to register in grid service on startup.
    -->
    <task id="1">
      <!-- Task unit factory. -->
      <factory>
        <ioc uid="my.unit.factory" policy="new">
          <java class="MyGridTaskUnitFactory"/>
        </ioc>
      </factory>

      <!-- Task topology. -->
      <topology>
        <ioc uid="basic.topology" policy="new">
          <java class="com.fitechlabs.xtier.services.grid.topology.GridTaskBasicTopology">
            <ctor>
              <arg null="true"/> <!-- No cluster group used. -->
              <arg type="float32">0.0</arg> <!-- No CPU-usage treshold. -->
              <arg type="boolean">true</arg> <!-- Allow local node. -->
              <arg type="boolean">true</arg> <!-- Allow local host nodes. -->
              <arg type="boolean">true</arg> <!-- Allow remote nodes. -->
            </ctor>
          </java>
        </ioc>
      </topology>

      <!-- Task router. -->
      <router>
```

```
<ioc uid="weight.router" policy="new">
  <java class="com.fitechlabs.xtier.services.grid.routers.GridTaskWeightRouter"/>
</ioc>
</router>

<!-- Failover resolver.-->
<failover>
  <ioc uid="fail.slow.resolver" policy="new">
    <java class="com.fitechlabs.xtier.services.grid.failover.GridTaskFailSlowResolver"/>
  </ioc>
</failover>
</task>
</region>
</xtier-grid>
```

Generally grid service configuration consists of one **<config>** XML tag which defines the common parameters for the grid and also includes an optional list of grid tasks specified by **<tasks>** XML tags. The **<config>** XML tag consists of the following elements:

- **ip-port** - This element specifies the IP port that is used by each grid node in the cluster. Note that you can either specify a default value using the default attribute or provide an IP port for a specific cluster node. Note also, that if you specify an IP port per cluster node, then the same port must be specified on each grid node for the same cluster node.
- **thread-pool-name** - This property specifies the name of the thread pool that is used for task unit processing internally in the grid service. The named thread pool must be specified in the Object Pool Service. (See the documentation for the **ObjectPoolService** for details on object pool configuration.)
- **max-exec-traces** - This property specifies the maximum number of grid tasks for which execution traces will be kept on this node. Note that if the limit is reached the oldest execution traces will be dropped.
- **taxonomy** - This optional element defines the IoC object that represents the grid taxonomy. The grid taxonomy defines the grid node's relative characteristics such as CPU weight, IO weight and memory weight. (See the documentation for the **IoCService** for more details on IoC usage.) If a taxonomy is not specified all grid nodes are considered identical as far as their weights.

The **<task>** XML tag has the following attribute and elements:

- **id** - This attribute defines the grid task ID. This ID is used throughout the grid service to identify a grid task.
- **Topology** - This element defines the IoC object that represents the topology resolver for this grid task. (See the documentation for the **IoCService** for more details on IoC usage.) The following section details how the topology is used in the grid service.
- **Factory** - This element defines the IoC object that represents the grid task unit factory for this grid task. (See the documentation for the **IoCService** for more details on IoC usage.) The following section details how the grid task unit factory is used in the grid service.
- **Router** - This element defines the IoC object that represents the task unit router for this grid task. (See the documentation for the **IoCService** for more details on IoC usage.) The following section details how the grid task unit router is used in 'grid' service.

- **Failover** - This element defines the IoC object that represents the failover resolver for this grid task. (See the documentation for the **IoCService** for more details on IoC usage.) The following section details how the failover resolver is used in the grid service.

## Using the xTier™ Grid Service

Let's consider the following task as an example: provide a weekly total report that requires resource intensive calculated financial indices for each weekday. The report will be requested by the end-user and will be generated dynamically in response. As such, the report must be generated in the minimum time possible.

Conceptually, this task can be split into separate calculations, to be performed in parallel, for each week day.

Please Note! The following code fragments given are simplified as much as possible. Some checks and synchronization considerations that are necessary in real applications are omitted for clarity of the example. This is done to increase code readability and to concentrate only on application logic.

## Mandatory Grid Elements

The first step is to implement the financial index calculation algorithm for split unit, i.e. one day in this case.

```
public class DailyReportCalculator {
    // Calculate financial measure for given day.
    public static double getFinancialMeasure(Date day) {
        double result;

        // Perform calculations.
        // ...

        return result;
    }
}
```

The preceding class performs financial index calculations for a given day.

Next, implement all mandatory grid service elements necessary for solving our task. The first mandatory grid element is the implementation of the **GridTaskUnit** interface, extending the convenient abstract class **GridTaskUnitAdapter**.

```
public class ExampleTaskUnit extends GridTaskUnitAdapter {
    /** Task unit ID of a "leave" task unit. */
    private static int SUB_TASK_UNIT_ID = 2;

    /** Auxiliary value, we will use it for date processing. */
    private Calendar calendar = Calendar.getInstance();

    /**
```

```
* Creates new example task unit with given parameters.
*/
ExampleTaskUnit(int tid, int uid, long eid) {
    super(tid, uid, eid);
}

/**
 * @see GridTaskUnit#aggregate(Set)
 */
public GridTaskUnitResult aggregate(Set results) {
    // Container for every day data.
    Map reportMap = new HashMap();

    // Iterate through results from all used grid nodes.
    for (Iterator iter = results.iterator(); iter.hasNext() == true;) {
        MarshalObject obj = (MarshalObject)((GridTaskUnitResult)iter.next()).getReturnValue();

        Date day = (Date)obj.getBoxedNotNull("day");
        double financialMeasure = obj.getFloat64("value");

        // Place day data into container.
        reportMap.put(day, new Double(financialMeasure));
    }

    MarshalObject ret = new MarshalObject();

    // Place container in MarshalObject.
    ret.putMap("reportMap", reportMap);

    return new GridTaskUnitResultAdapter(GridTaskUnitResult.TASK_UNIT_OK, getTaskId(),
        getUnitId(), getExecId(), ret);
}

/**
 * @see GridTaskUnit#exec(Marshallable)
 */
public GridTaskUnitResult exec(Marshallable arg) {
    MarshalObject argObj = (MarshalObject)arg;

    MarshalObject retObj = new MarshalObject();

    // Gets task input parameter - day of report calculation.
    Date day = (Date)argObj.getBoxedNotNull("day");

    // Calculate financial measure for given day.
    double financialMeasure = DailyReportCalculator.getFinancialMeasure(day);

    // Place argument and result into MarshalObject.
```

```
// Note: we have to save day to know to which day calculated financial index concerns
retObj.putDate("day", day);
retObj.putFloat64("value", financialMeasure);

return new GridTaskUnitResultAdapter(GridTaskUnitResult.TASK_UNIT_OK, getTaskId(),
    getUnitId(), getExecId(), retObj);
}

/**
 * @see GridTaskUnit#split(Set, GridTaxonomy, Marshallable)
 */
public Set split(Set grid, GridTaxonomy tax, Marshallable arg) {
    MarshalObject argObj = (MarshalObject)arg;

    // For simplicity we use one-level split here
    // If task is a subtask, further split does not performed.
    // This given time interval is split by days only.
    if (getUnitId() == SUB_TASK_UNIT_ID) {
        return null;
    }

    // Get time interval for calculations, specified by user.
    Date dateFrom = (Date)argObj.getBoxedNotNull("dateFrom");
    Date dateTo = (Date)argObj.getBoxedNotNull("dateTo");

    calendar.setTimeInMillis(dateFrom.getTime());

    // Calculate number of days in the given interval.
    int days = (int)(dateTo.getTime() - dateFrom.getTime()) / (1000 * 60 * 60 * 24);

    Set refs = new HashSet();

    // Split task for per-day computations.
    for (int i = 0; i < days; i++) {
        MarshalObject obj = new MarshalObject();

        calendar.add(Calendar.DAY_OF_WEEK, i);

        // Pass calculation date as argument.
        obj.putDate("day", calendar.getTime());

        // Add task unit references to returning set of references.
        refs.add(new GridTaskSplitRefAdapter(getTaskId(), SUB_TASK_UNIT_ID, getExecId(), obj));
    }

    return refs;
}
}
```

There are several method implementations in this code to consider:

- **split()** - This method splits the task unit into grid nodes. The method can return **null** to reject the split and indicate intent to execute task unit locally. In other cases method should return a set of **GridTaskSplitRef** instances. In this example there is only one split where the task is split into single days.
- **exec()** - This method executes the task unit on a certain grid node. This method is called when the service determines that a given task unit should be executed locally (when **split()** returned null or the topology contains only the local node). This example has only the calculation day as a single argument, and the function to make the financial calculations is called.
- **aggregate()** - For every task unit that is split into sub tasks this method forms the task unit result based on the results of execution of all the subtasks of this task unit. On the level of root task unit the final result of the whole task execution is formed. In this example the results of the computations for each day are combined and returned as there is only one level of splitting with this implementation.

The next mandatory step is to implement the GridTaskUnitFactory interface

```
public class ExampleGridTaskUnitFactory implements GridTaskUnitFactory {  
    /**  
     * @see GridTaskUnitFactory#newTaskUnit(int, int, long)  
     */  
    public GridTaskUnit newTaskUnit(int tid, int uid, long eid) {  
        /**  
         * Since in our example all work units represented by the same class  
         * we ignore grid task, execution and unit IDs.  
         * -----  
         */  
        return new ExampleTaskUnit(tid, uid, eid);  
    }  
}
```

Please note that for a single task “task units” can be implemented by several different classes when, for example, performing calculations on different levels. In this example all calculation logic for all nesting levels is encapsulated in a single class. Thus while creating instances of **ExampleTaskUnit** we don’t use grid task, execution and task unit IDs.

The following is the xTier™ Grid Service configuration file fragment relevant to this example.

```
<!--  
  Grid task to register in grid service on startup.  
  Example .  
-->  
<task id="1">  
  <!-- Task unit factory. -->  
  <factory>  
    <ioc uid="example.unit.factory" policy="new">  
      <java class="ExampleGridTaskUnitFactory"/>  
    </ioc>
```

```
</factory>
...
</task>
```

In this example we only define the **ExampleGridTaskUnitFactory**. (For information about setting other XML parameters – refer to the section that discusses “Service configuration”.)

The final step is to use the xTier™ Grid Service to solve the task.

```
...
// Unique task identifier.
private static final int EXAMPLE_TID = 1;

// Get the instance of xTier kernel.
XtierKernel xtier = XtierKernel.getInstance();

...
// Create grid task argument.
MarshalObject arg = new MarshalObject();

// dateFrom / dateTo - time interval for report.
arg.putDate("dateFrom", dateFrom);
arg.putDate("dateTo", dateTo);

// Get the XML-defined grid task.
GridTask task = grid.getTask(EXAMPLE_TID);

// Execute grid task.
GridTaskResult result = grid.exec(task, arg);

if (result.isSuccessful() == true) {
    MarshalObject retval = (MarshalObject)result.getReturnValue();

    Map reportMap = retval.getMap("reportMap");

    for (Iterator iter = reportMap.keySet().iterator(); iter.hasNext() == true;) {
        Date day = (Date)iter.next();

        double financialMeasure = ((Double)reportMap.get(day)).doubleValue();

        // Print out result by days.
        System.out.println("Day = " + day + ", financialMeasure = " + financialMeasure)
    }
} else {
    // Handle error result...
}
```

As demonstrated by this example, the xTier™ Grid Service provides a straight-forward, and powerful facility to utilize grid technologies for distributed tasks. The developer needs to implement only two special classes while the rest of the functionality and the rest of functionality such as:

- All topology management
- Failover resolution
- Tracking timeouts
- Execution tracing
- On-demand resource provisioning

will be executed by the xTier™ Grid service.

### Optional Grid Elements

In most cases it should be enough for the developer to use the predefined classes implementing the topology, router and failover resolver. In some special cases, the developer may be required to implement these components themselves. The following are some examples of custom implementations of optional elements that can be implemented by developer:

#### GridTaskTopology – defines grid system topology

```
public class CustomTopology implements GridTaskTopology{
    /**
     * @see GridTaskTopology#getNodes(int, int, long)
     */
    public Set getNodes(int tid, int uid, long eid) {
        // Getting cluster service instance.
        ClusterService cluster = XtierKernel.getInstance().cluster();

        // Getting all active nodes.
        Set allNodes = cluster.getAllNodes();

        Set gridNodes = new HashSet();

        for (Iterator iter = allNodes.iterator(); iter.hasNext() == true;) {
            ClusterNode node = (ClusterNode)iter.next();

            if (node.getNumberOfCpus() >= 2) {
                gridNodes.add(node);
            }
        }

        return gridNodes;
    }
}
```

In the preceding example, only nodes having 2 or more CPU's are included into grid.

### **GridTaskRouter – defines a custom task router.**

```
public class CustomRouter implements GridTaskRouter {
    /** */
    private Random random = new Random();

    /**
     * @see GridTaskRouter#route(GridTaskSplitRef, Set, GridTaxonomy)
     */
    public ClusterNode route(GridTaskSplitRef ref, Set topology, GridTaxonomy tax) {
        int nodesCount = topology.size();

        ClusterNode[] gridNodes = (ClusterNode[])topology.toArray(new ClusterNode[nodesCount]);

        int nextNode = random.nextInt(nodesCount);

        return gridNodes[nextNode];
    }
}
```

In this example the grid node for the next task unit execution is chosen randomly from the set provided by the topology.

### **GridTaskFailoverResolver – define the failover resolver.**

```
public class CustomFailoverResolver implements GridTaskFailoverResolver {
    /**
     * @see GridTaskFailoverResolver#resolve(GridTaskSplitRef, GridTaskUnitResult)
     */
    public int resolve(GridTaskSplitRef ref, GridTaskUnitResult res) {
        switch (res.getReturnCode()) {
            case GridTaskUnitResult.TASK_UNIT_OK: {
                // Should never reach.
                return 0;
            }

            // Terminal state.
            case GridTaskUnitResult.ERR_NO_RETRY:
            case GridTaskUnitResult.ERR_INVALID_ARGS:
            case GridTaskUnitResult.ERR_TASK_UNIT_TIMEOUT:
            case GridTaskUnitResult.ERR_SYSTEM_FAILURE:
            case GridTaskUnitResult.ERR_TASK_UNIT_NOT_FOUND:
            case GridTaskUnitResult.ERR_NOT_EXEC_TASK_UNIT:
            case GridTaskUnitResult.ERR_TASK_NOT_FOUND:
            case GridTaskUnitResult.ERR_RETRY_OTHER:
            case GridTaskUnitResult.ERR_RETRY_SAME: {
```

```
        return GridTaskFailoverResolver.TASK_UNIT_TERMINATE;
    }

    // No support for user-defined error code.
    case GridTaskUnitResult.ERR_USER_DEFINED: {
        System.err.println("Some text...");

        return GridTaskFailoverResolver.TASK_UNIT_TERMINATE;
    }

    default: {
        // Should never reach.
        return 0;
    }
}
}
```

In the given example execution will be stopped independently of GridTaskUnitResult return code. For user defined error (GridTaskUnitResult.ERR\_USER\_DEFINED) an additional message will be printed to error console.

### GridTaxonomy – defines a custom taxonomy

Assume that the relative CPU weights for the grid cluster is to be defined. The following is an example of the XML configuration required to do this.

```
<ioc policy="new" uid="example.taxonomy">
  <java class="ExampleGridTaxonomy">
    <call method="addNode">
      <arg type="string">machine-1</arg>
      <arg type="int32">1</arg>
    </call>
    <call method="addNode">
      <arg type="string">machine-2</arg>
      <arg type="int32">3</arg>
    </call>
    ...
  </java>
</ioc>
```

Please Note:

- "machine-1" & "machine-2" are host names of grid cluster machines.
- With respect to CPU weight "machine-2" has a bigger (3 times) priority to receive tasks when they are distributed than "machine-1".

```
/**
 * Simple example of custom taxonomy implementation.
 * This taxonomy implementation takes into account only CPU utilization ratio.
 */
public class ExampleGridTaxonomy implements GridTaxonomy {
    /** No information. */
    private final static int NO_INFO = -1;

    /** List of nodes with CPU weights. */
    private Map nodes = new HashMap();

    /**
     * Method for filling IoC object from XML.
     */
    public void addNode(String hostName, int cpuWeight) {
        nodes.put(hostName, new Integer(cpuWeight));
    }

    /**
     * @see GridTaxonomy#getCpuWeight(ClusterNode)
     */
    public int getCpuWeight(ClusterNode node) {
        Integer cpuWeight = (Integer)nodes.get(node.getAddress().getHostName());

        // if node is not defined return NO_INFO.
        return (cpuWeight == null ? NO_INFO : cpuWeight.intValue());
    }

    /**
     * @see GridTaxonomy#getIoWeight(ClusterNode)
     */
    public int getIoWeight(ClusterNode node) {
        return NO_INFO;
    }

    /**
     * @see GridTaxonomy#getMemoryWeight(ClusterNode)
     */
    public int getMemoryWeight(ClusterNode node) {
        return NO_INFO;
    }
}
```

Instances of this class are created by the service as IoC objects (the addNode() method reads data from XML file). The getCpuWeight() method returns the user-defined CPU weight values.

## Conclusion

The basic aspects of grid technology were described in this article, a classification of grid systems by purpose and by audience was made, and examples of the grid systems of various types from the major vendors of grid software were given. Finally the xTier™ Grid Service was introduced and explained along with a simple code-level example. Given the example, it is easy to understand how the xTier™ Grid Service's characteristics of simplicity and high flexibility provide the developer with an easy to use "toolkit" to develop nRT Computational Grids. These features make xTier™ unique in the marketplace when compared to the implementations chosen by the other grid vendors.

With the xTier™ Grid Service:

- The developer has a powerful tool to solve a variety of industrial and business tasks.
- Developers can decrease execution time for many types of tasks and applications, using the xTier™ Grid Service.
- Developers can develop new functionality that could not have been developed due to extended execution time without using the xTier™ Grid Service.

The xTier™ Grid Service brings grid technology to the enterprise application developer as with the xTier™ Grid Service, grid implementations are no longer only the prerogative of academic research and organizations that can afford multi-million dollar SLA agreements. No, using the xTier™ Grid Server, every developer can use the power of grid implementations in a cost-effective and highly performant way.

### Fitech Laboratories Inc.

#### Corporate Headquarters

Century Plaza, Suite 405  
1065 East Hillsdale Boulevard  
Foster City, CA 94404  
USA

Phone: 1-650-578-0808  
Fax: 1-650-578-0805

#### East Coast Sales Office

330 Madison Ave., 9th floor  
New York, NY 10017  
USA

Phone: 1-646-495-5076

#### Fitech Laboratories Japan

Toranomon40 MT Bldg. 3F  
5-13-1 Toranomon Minato-ku  
Tokyo 105-0001, Japan

Phone: +81-3-5402-7711  
Web: [www.fitechlabs.co.jp](http://www.fitechlabs.co.jp)