A CASE STUDY OF MTTS PROJECT

First Report
8 July 2010

Author
Ijaz Ahmed

Under the supervision of
Prof. Nestor Catano

Aeminum Project
CMU | Portugal Partnership
University of Madeira

Contents

1 Introduction: 1

2 Scope 2

3 Objectives 2

4 Organization (Filtering) of Code 2

5 System Views: 3

5.1 Client-Server View: 3

5.2 Task Registration and Controller View: 4

5.3 Task Execution View: 5

5.4 Database View: 6

5.4.1 mtts-queue: 6

5.4.2 queueTable: 7

5.4.3 dependencyTable: 7

6 System Description: 8

6.1 Task: 8

6.1.1 Task Registration: 8

6.1.2 Task Status: 8

6.1.3 Frozen Task: 8

6.1.4 Task Selection: 8

6.2 Queues and Threads: 9

6.3 Concurrency: 9

6.4 System Working: 10

6.4.1 Server Responsibility: 10

6.4.2 Client Responsibility: 10

6.4.3 An Example: 10

7 Packages and Classes View: 12

7.1 mtts Package: 13

7.1.1 mtts-api Package: 14

7.1.1.1 mtts-api Classes: 15

7.1.2 server Package: 16

7.1.2.1 server Package Classes: 16

7.2 il (Intelligent Lock) Package (from Incubator): 17

7.2.1 il (Intelligent Lock) Package Classes: 18

7.3 Model View Controller of MTTS: 19

8 Related Tools and Techniques: 21

8.1 TypeStates and Access Permissions: 21

8.2 Plural: 23

8.3 Plaid: 24

9 Work Plan: 24

9.1 Targeted Properties: 24

9.2 Time Plan: 25

10 APENDIX: 26

10.1 auxtestlib Package: 26

10.1.1 auxtestlib Package Classes: 26

10.2 Incubator Package: 27

10.2.1 appstarter Package: 28

10.2.2 bof package: 28

10.2.3 pool package: 28

10.2.3.1 pool Package Classes: 29

10.2.4 ui Package: 30

10.2.4.1 ui Package Classes: 30

10.2.5 doctype Package: 31

10.2.5.1 doctype Package Classes: 31

10.2.6 dProber Package: 32

10.2.6.1 dProber Classes: 32

10.2.7 dyncl Package: 33

10.2.7.1 dyncl Package Classes: 34

10.2.8 Ui Package: 34

10.2.8.1 Ui Package Classes: 35

10.2.9 rmi Package: 35

10.2.9.1 rmi Package Classes: 35

10.2.10 Ui Package: 36

10.3 relogger Package: 36

10.3.1 relogger Package Classes: 36

10.4 Model View Controller Architecture of nonMttsCode: 38

1 Introduction:

The Multi-Task Thread Server is an industrial application provided by Novabase, AEminium's industrial partner. Multi-Task Thread Server (MTTS) is the core implementation of a task distribution and organization server. This core is used to gain parallelism and high availability by running tasks among different threads. It can be used to process various documents in parallel. It can also be used to migrate documents among different repositories (e.g. doing data backup). The core can support a high volume of tasks. The application is currently being used in the financial sector to archive (manipulate) bank checks with usually loosely time bound tasks.

The MTTS considers task as a basic processing activity. The definition of task is quite generic. Task stores information related with identification, dependencies and priority etc. From the documentation of the MTTS application and discussions with Novabase Engineers who wrote the application, it appears evident that the tasks do not have a complex dependencies structure. Usually most of the tasks are independent with the exception of few tasks; however the MTTS core has the ability to execute the tasks with complex dependencies structure. In this write-up, the words "*MTTS core*", "*MTTS server"* and *"MTTS"* are synonyms.

The MTTS core can be considered as an engine as it provides the mechanism to distribute the tasks on a parallel basis. The MTTS only provides the mechanism to distribute the tasks and it does not execute them actually. Once the mechanism to distribute the tasks is defined, the rest of the process is straight-forward (just a single routine is required to complete the execution). The execution of a task depends on its nature, e.g. if the task is to load the documents, then the implementation would contain the code to load the documents and if the task is to archive the documents, then the implementation would contain the code to archive the documents. The MTTS core can also be considered as a general purpose library to execute generic tasks in parallel; the core makes no assumption on the nature of these tasks.

The MTTS server uses queues to execute tasks. Nonetheless, a single task can be added to only one queue. Now, every queue has a number of threads to execute the tasks. Each queue can have a different number of threads according to the MTTS configuration. However, the number of threads associated to a queue remains fixed and the administrator of the MTTS can increase/decrease the number of running (working) threads according to the work load per queue and the nature (urgency) of the tasks in special circumstances. The MTTS server has the ability to perform general functionalities (usually associated with a task distribution environment) such as add, delete and retry tasks.

2 Scope

The MTTS core receives different tasks from clients and distributes those tasks among different threads to complete the tasks as early as possible. The MTTS core does not execute those tasks; Nonetheless the task execution is fairly simple and dependant on nature of task and can be implemented accordingly.

3 Objectives

The main objectives of this case study are

1. To demonstrate the usefulness and effectiveness of the tools and techniques developed in the Project AEminium in the specification and verification of a commercial distributed application. We are interested in verification the absence of general concurrent problems, e.g. race-conditions and deadlocks, as well as general programming problems, e.g., null referencing or accessing a structure out of its bounds.
2. To demonstrate the usefulness of the tools and techniques developed in AEminium in the improvement of a distributed application. We are interested in parallelizing the main tasks of the application so as to improve processing times.

4 Organization (Filtering) of Code

The code (that Novabase provided) can be divided into 3 subgroups, namely "*ProcessingCode*","

GUICode" and *"TestcaseCode".* The "*TestcaseCode*" is related with test cases and "*GUICode*", which is related with user interface. The "*ProcessingCode"* can be further subdivided into two main groups, "*MTTS-ProcessingCode"* and *"nonMTTS-ProcessingCode"*. The "*MTTS-ProcessingCode"* is related with the MTTS core functionalities whereas *"nonMTTS-ProcessingCode"* is being used by some other applications of the Novabase. This report provides details of "*MTTS-ProcessingCode"*, however the brief details of the *nonMTTS-ProcessingCode* are added to the appendix.

5 System Views:

The following section describes different views of the MTTS application. The client-server view provides the conventional client-server view of the system, whereas the task-registration and controller view provides some insights that how tasks are registered and controlled. The last view exposes some more fine details of the system.

5.1 Client-Server View:

The figure1 presents the simplest view of the MTTS. First of all, several instances of the MTTS can coexist in a typical configuration and several clients can request services from various servers. These services are related to the execution of tasks. Every MTTS publishes its RMI interfaces so that the clients can connect to any of the available MTTS by using these interfaces. The clients request MTTS for task registration and the MTTS stores those tasks in the database. After that, the MTTS fetches the tasks and the worker threads of MTTS execute those tasks. The MTTS do not interact with each other directly and every server has its own database; however all the databases are associated with the main database. The MTTS maintains a separate database for each of its queue. We will discuss queues and its role in the forthcoming sections.

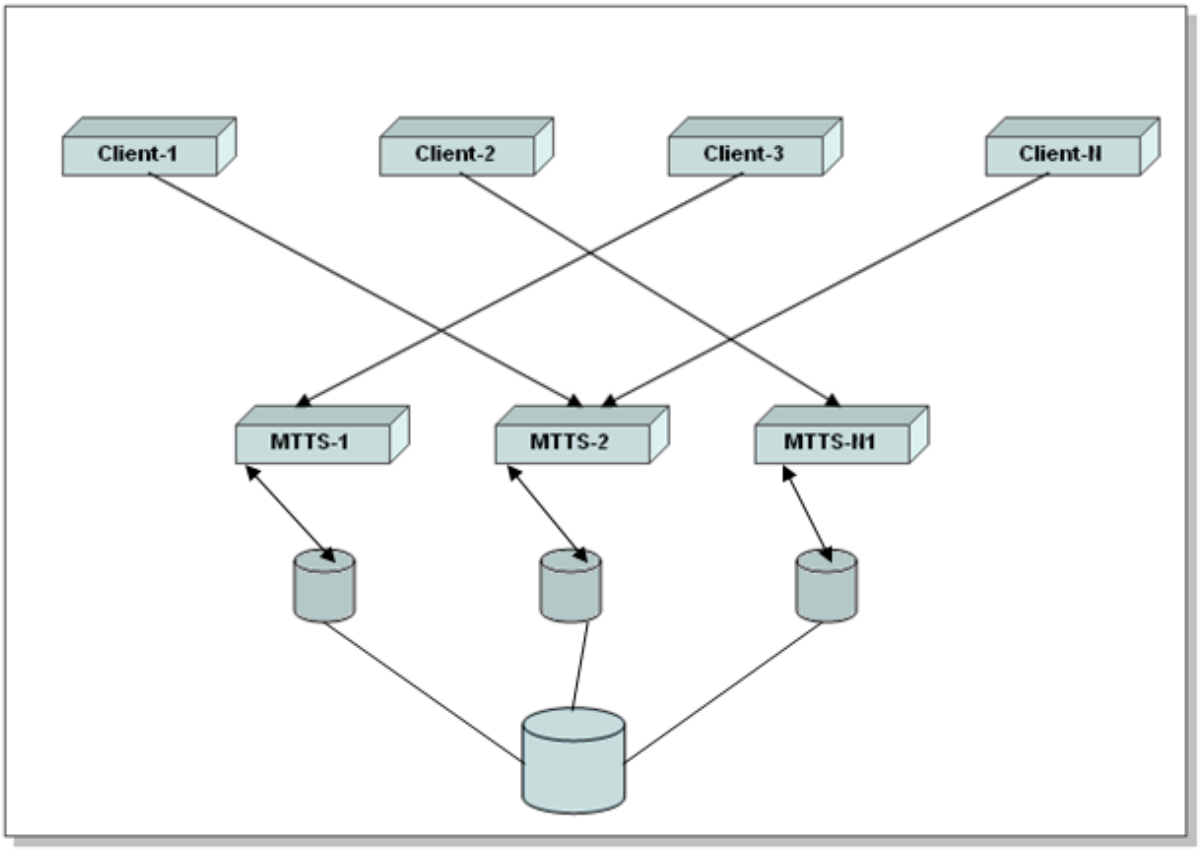


Figure 1: Client Server View

5.2 Task Registration and Controller View:

Figure2 shows the Task Registration and Controller View. The MTTS has 4 main components: a RMI (Runtime Method Invocation) component, a task registration component, a remote operation control and a thread pool. The RMI component provides an interface to clients for communication. The clients use RMI to register their tasks with the task registration component. During tasks registration, the tasks are added to the server database for execution. The remote operation control is used to check and control the task progress. The thread pool contains a controller and several worker threads. The controller is used to control the worker threads which fetch tasks from the database and execute them. The worker threads report back the status of the tasks to the controller.

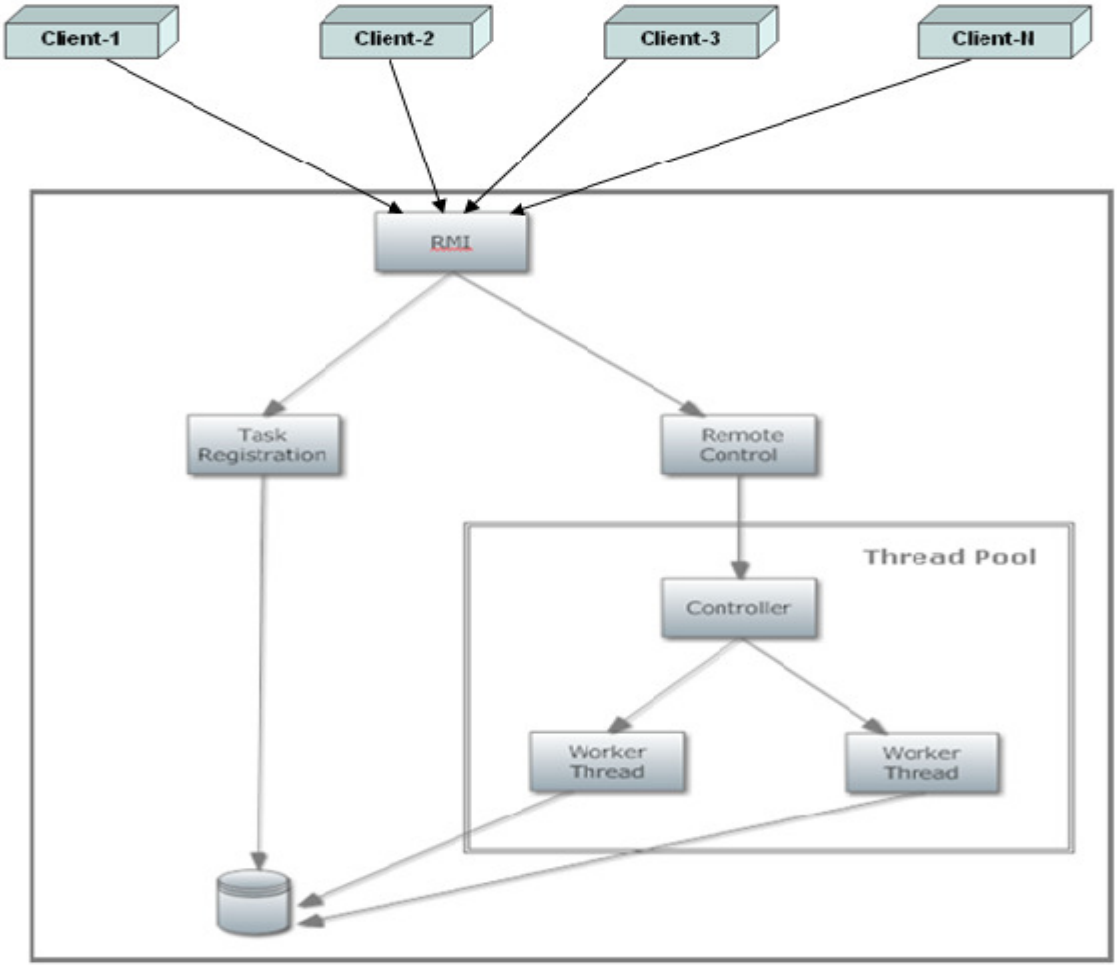


Figure 2: Task Registration and Controller View

5.3 Task Execution View:

Figure3 shows the details about task queues and threads synchronization. The server creates task queues to manage the tasks. A server can create more than one queue according to its configuration. Queues have working threads to fetch and execute the tasks along with the ability to synchronize those threads with the help of a mutex key algorithm. The queue can have a different number of threads according to its configuration and every queue has its own database to store tasks. When a queue is created, its associated database is also created. Every queue worker threads have unique repository to fetch tasks. The threads of one queue cannot access the tasks (database) of other queues belonging to the same server or a different server.

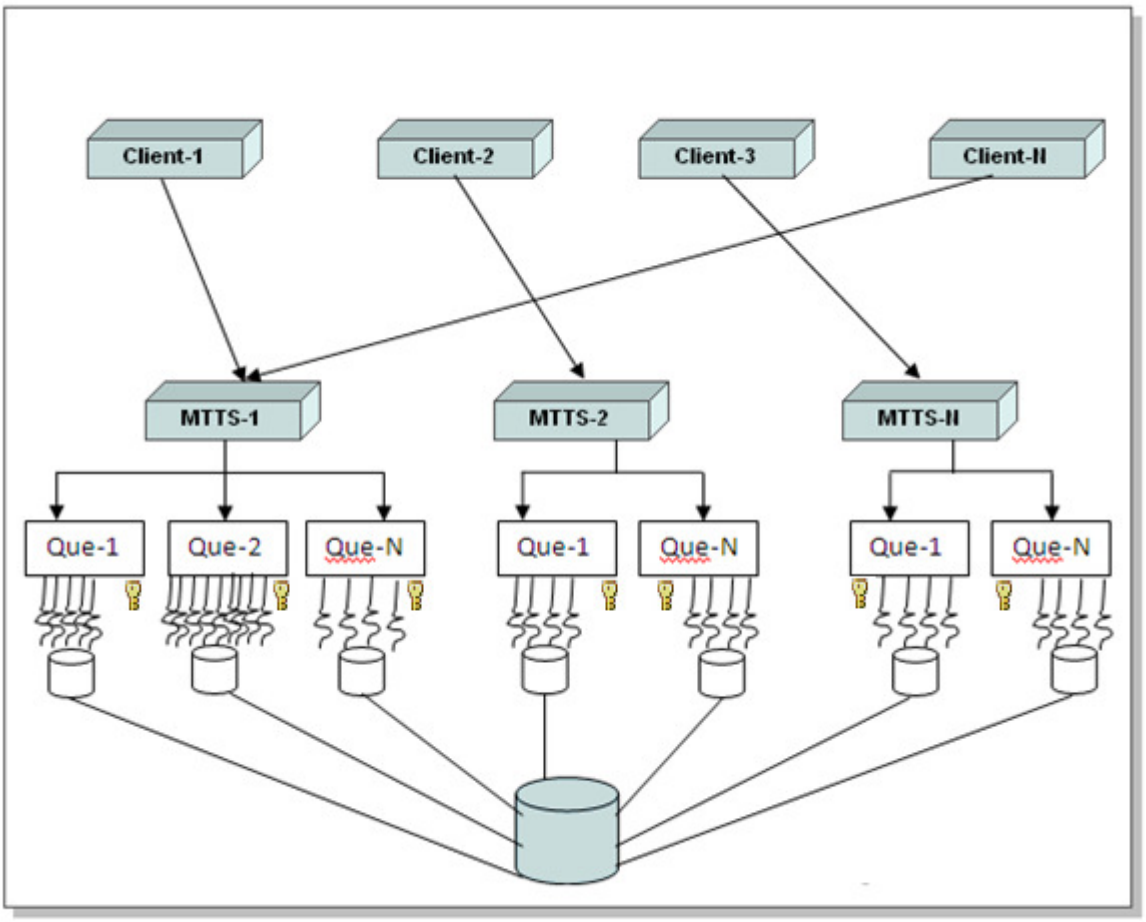


Figure 3: Task Execution View

5.4 Database View:

This section describes the MTTS database view. The database is used to store the configuration information of the server and the details of the task. The task details are related with task identification, task priorities and dependencies etc.

5.4.1 mtts-queue:

This table is a configuration table and is used by the MTTS server to create queues and working threads. The table has fields *name*, *description*, *ithreads* and *maxthreads*. The meaning of fields *name* and *description* is obvious whereas *ithreads* indicates the initial number of worker threads and *maxthreads* contains the maximum number of worker threads. At the startup of

the server, the queues have an initial number of threads, whereas the administrator can increase the number of threads but not greater than *maxthreads*. The server can have any number of queues according to its configuration.

name	ithreads	maxthreads	Description
Que1	5	10	This queue is related with document loading
Que2	10	20	This queue is related with document archiving.
Que3	100	150	This queue is related with backup tasks.

5.4.2 queueTable:

This table is used to store the tasks and their data. This table has columns *id, name, description, priority, class, tslock, haserror* and *startt*. The meaning of *id*, *name*, *description* and *priority* is obvious from the meaning of the word whereas *tslock* means task is being processed and *haserror* means the task has some problems during execution. The column *startt* has the registration time whereas the column *class* stores the name of the class that will actually perform the task. The tasks are added in this table and later fetched for execution.

id	Name	description	priority	class	tslock	haserror	startt
1	T1	------	1	Load.java	0	0	10:53
2	T2	------	2	Load.java	0	0	10:53
3	T3	--------	2	Load.java	0	0	10:53
4	T4	-------	4	Load.java	0	0	10:53

5.4.3 dependencyTable:

This table describes the dependency information. The table has two columns *id* and *did*. The id means identification of task where as *did* means the identification of task whom *id* is dependent. It is worth to mention here that the task dependencies are provided by client. The below snapshot states that *Task1* depends on *Task5* and *Task3* depends on *Task4*. Therefore, if a task is independent then its *id* does not appear in this table.

Id	did
1	5
3	4

6 System Description:

This section provides the details of the concepts that MTTS uses. It defines the concepts related with task, queues and concurrency. This section also describes the working of the system with the help of an example.

6.1 Task:

A task is a basic activity that the MTTS server has to perform. The task has properties such as *id, name, priorities, starting time* etc along with the ability to execute itself.

6.1.1 Task Registration:

The client requests the server to register the task by using the RMI interface. The server adds that task in the database. It is the responsibility of the client to identify the task priority and its dependencies along with the other related information. However the server will assign identification (id) to the task along with some other information, such as registration time.

6.1.2 Task Status:

The task has different status such as completed, not completed and error. Moreover if some problem occurs during the execution of a task then it would be treated as incomplete. There is no concept of partial completion of a task.

6.1.3 Frozen Task:

A task that is not executed from a long time due to any reasons is considered as a frozen task.

6.1.4 Task Selection:

The MTTS server uses a dynamic mechanism to select the tasks from the database for

execution. The mechanism states that "The tasks that have no dependency (or whose dependencies have been completed) and have high priority should be executed first". Suppose there are 3 tasks namely *T1*, *T2* and *T3* and their dependency information is as follows, *T1* depends on *T2* and *T2* depends on *T3* and *T3* does not depend on anything. Moreover *T1* has higher priority than *T2* and *T2* has a higher priority than *T3*. Now the server first executes (fetches) task T3, as it does not depend on any other task and ignores the other tasks that have even higher priorities than this task. After the completion of T3, the server executes T2 and then T1.

6.2 Queues and Threads:

The MTTS server generates queues and then associates worker threads with each queue. The information about number of queues and associated threads is stored in the configuration table. It is possible that a MTTS server creates more than one queue, e.g. it creates 3 queues, namely *Que1*, *Que2* and *Que3*. The first queue *Que1* might have worker threads that perform the document loading task, the second queue *Que2* might have worker threads that perform document archiving tasks whereas the third queue *Que3* threads might deal with the backup task. The queues create some threads initially; however the administrator can modify the number of threads according to the prevailing requirements. Each queue can have different numbers of threads, such as *Que1* has 10 threads, *Que2* has 20 threads and *Que3* has 30 threads and if the administrator feels that there is some urgency in document loading task, so he/she may increase the number of threads accordingly for queue *Que1*.

6.3 Concurrency:

The MTTS server uses threads to execute task concurrently. The execution of tasks is mutual exclusive and is ensured by a mutual exclusion algorithm (mutex). There are a number of situations in which threads require synchronization. The following scenario explains one of the possible situations at abstract.

Suppose queue *Que1* has 10 threads and the work is to load the 10000 documents. When thread *Thread1* is fetching the tasks from database to perform that work, no other thread can

fetch the same task from the database; hence the same task is not executed simultaneously by the two threads. The MTTS server uses a mutex algorithm to create a critical section, so that only one thread that acquires the lock on the task first will be able to process the task. The MTTS server is also storing the thread status (such as sleeping, executing, fetching etc) and statistics (number of threads) to keep an eye over the threads.

6.4 System Working:

This section highlights the major responsibilities of server and client with brevity. This section also provides an example to explain, how the system works.

6.4.1 Server Responsibility:

Following are the major responsibilities of the server.

1- The server provides an RMI interface to clients so that they may establish connection with server.
2- The server provides an RMI interface to clients so that they may register tasks with the server.
3- The server fetches the tasks from the database and distributes those tasks among threads for execution.
4- The server adds some information to the database for logging purpose.

6.4.2 Client Responsibility:

Following are the major responsibilities of the client.

1- The client has to search the MTTS server and requests for registration.
2- The client has to define the tasks along with their priorities and dependencies.
3- The client may request the server to register its defined tasks for execution.

6.4.3 An Example:

The MTTS server reads the configuration information from the database table *mtts-queue* and

creates 3 queues along with the initial number of threads (*ithreads*).

mtts-queue

name	ithreads	maxthreads	description
Que1	5	10	This queue is related with document loading
Que2	8	20	This queue is related with document archiving.
Que3	16	150	This queue is related with backup tasks.

So we have a situation like this. Here | represents a thread that is ready to execute the task.

Que1 **Que2** **Que3**

{| | | | | } {| | | | | | | | } { | | | | | | | | | | | | | | | |}

Every queue creates its own database tables such as *queueTable and dependencyTable*. So we have a situation like this

Que1 **Que2** **Que3**

{| | | | | } {| | | | | | | | } { | | | | | | | | | | | | | | | |}

[queueTable, dependencyTable] [queueTable, dependencyTable] [queueTable, dependencyTable]

Now consider the following snapshot of *queueTable* and *dependencyTable* for *Que1*. We may ignore the *Que2* and *Que3* tables for the sake of brevity.

queueTable for Que1:

This table contains the information about tasks.

Id	name	description	priority	Class	tslock	haserror	startt
T1	Task1	------	1	Load.java	0	0	10:53
T2	Task2	------	2	Load.java	0	0	10:53
T3	Taske3	--------	2	Load.java	0	0	10:53
T4	Task4	-------	4	Load.java	0	0	10:53
T5	Task5	--------	4	Load.java	0	0	10:53
T6	Task6	--------	4	Load.java	0	0	10:53

.							
.							
.							
T1000	Task1000		1	Load.java	0	0	10:53

dependencyTable for Que1:

Id	Did
.	.
.	.
.	.
.	.
T1000	T990

Now the 5 threads of *Que1* are ready to execute jobs associated with their tables. The 5 threads are trying to fetch the jobs from the same repository, so the mutex algorithm is applied to ensure synchronization. The criteria of fetching the job is *"Select the task that have highest priority and no dependency (or whose dependency is being completed)".*

Now we have a situation like this

Que1

[T1]	[T30]	[T104]	[T29]	[T505]

The task T1 is allocated to first thread and the task T30 is allocated to the second thread and so on. Now every thread has a task to execute and once the task is completed, the threads will fetch new tasks from its *queueTable* for execution.

7 Packages and Classes View:

There are 4 main packages of the MTTS application, namely *auxtestlib*, *incubator*, *mtts* and *relogger*, as shown in the figure below. *auxtestlib* is related with testcases; *relogger* is related with logging facility (not used by MTTS); *incubator* is performing miscellaneous functionality

along with the ability to provide synchronization facility for threads. Not all the packages of *incubator* are used by MTTS application; however it is used for some other applications written by Novabase engineers. Novabase provided the MTTS application for this case study. This report discusses only those packages of *incubator* that are related with the MTTS functionality. *mtts* is the main package that provides the facility to distribute tasks among threads.

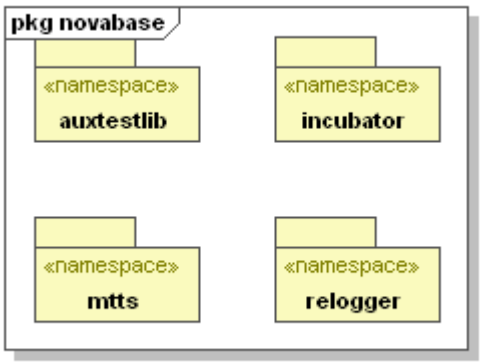


Figure 4: Packages

7.1 mtts Package:

This package is the core package of the MTTS application which provides an RMI facility to clients for connection. This package also provides facilities such as task definition, task registration and task distribution. This package also does necessary interaction with the database. It has further packages namely *mtts-api*, *server*, *mrrmisync* and *rmicomm*. *mtts-api* is related with task definition and execution whereas *server* is related with task registration and distribution. *mrmisync* contains only interfaces (that are not implemented) and *rmicomm* relates to RMI communication. Both *mrmisync* and *rmicomm* are not used by the MTTS; however the necessary functionality for RMI is implemented in the *mtts-api* package.

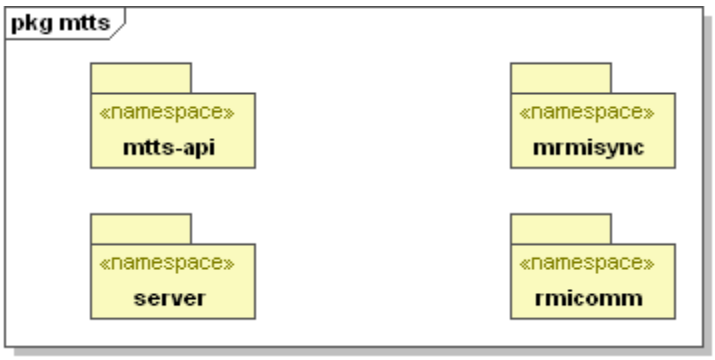


Figure 5: mtts Packages

7.1.1 mtts-api Package:

This package defines classes for the task and RMI interface for connection. The following diagram shows the basic class structure of the package.

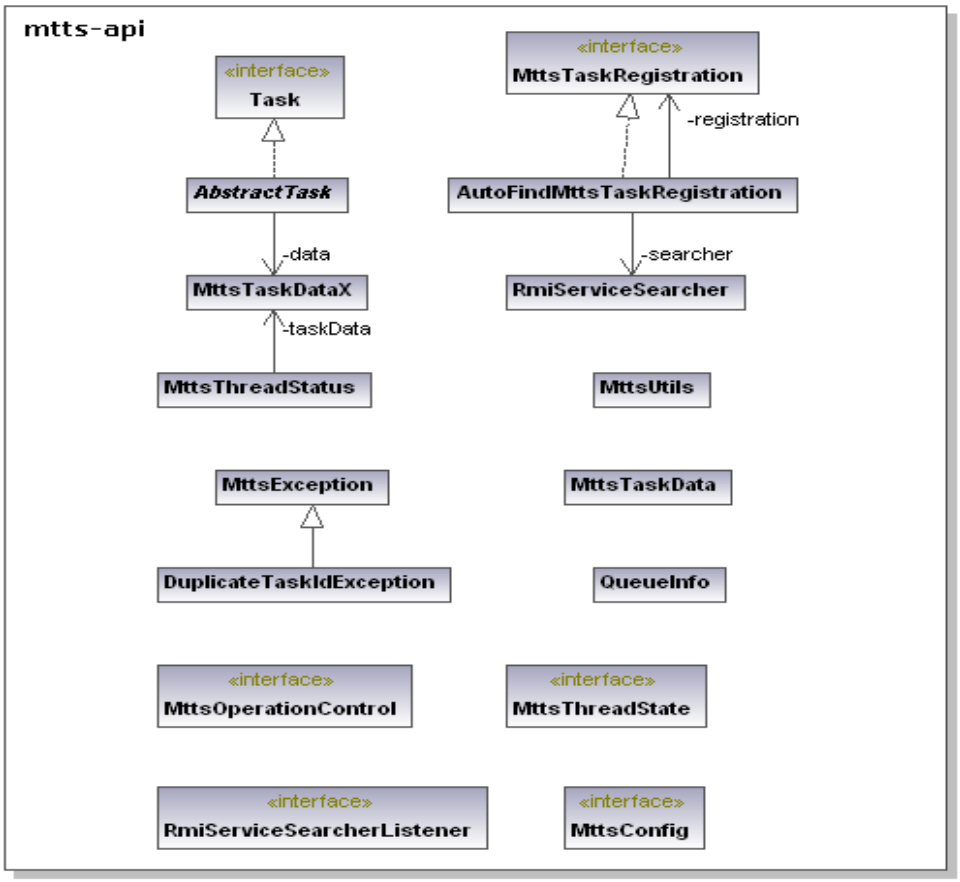


Figure 6: mtts-api Package Classes

7.1.1.1 mtts-api Classes:

This section provides the details of the classes.

Class	Details
`Task`	*This is interface for task. A task is a basic activity that the server is supposed to perform*
`AbstractTask`	*This is an abstract class and defines the task at abstract.*
`MttsTaskDataX`	*This class represents the information associated with a task. The task has attributes such as task id, task dependencies, starting time and status etc. This is a structure that stores all the necessary information regarding the task and it has a method execute the task.*
`MttsTaskRegistration`	*This is an interface to register tasks. The client requests server to register (add) its task prior to task execution.*
`AutoFindMttsTaskRegistration`	*This class finds a list of hosts and then connects to the available server (host). If the connection drops, it automatically tries other link. This class provides the facility to register tasks with the MTTS server.*
`RmiServiceSearcher`	*This class searches RMI services in one or more hosts to connect with the hosts.*
`RmiServiceSearcherListener`	*This interface is a Listener that is notified when a service (host) is found.*
`MttsConfig`	*This class stores some of the MTTS Configuration information such as database driver, user name and password.*
`MttsThreadState`	*This interface defines different thread states, such as sleeping, executing, fetching, etc.*
`MttsThreadStatus`	*This class stores different status information about threads such as how much work (task) is done (completed) by a thread.*
`MttsUtils`	*This class provides different utility functions such as to check the aliveness of server, list of the registered tasks,*

	and validate of port numbers.
`QueueInfo`	*This class provides basic information about queues such as name, the number of initial threads and the number of maximum threads in a queue.*
`MttsOperationControl`	*This interface is used to monitor the operation of MTTS. The interface is used to perform different operations such as retry frozen tasks, delete task, and do enquiry about tasks.*

7.1.2 server Package:

This package is responsible to distribute the tasks among threads. This package fetches the task from the database and put the task in a queue with the help of *QueueManager* and assigns threads to the queue. It creates the task with the help of *TaskCreator* and assigns to the *ExecutionThread* for further processing. This package also provides the facility to remotely view (manage) the task associated with the server with the help of class *RemoteOperationControl*.

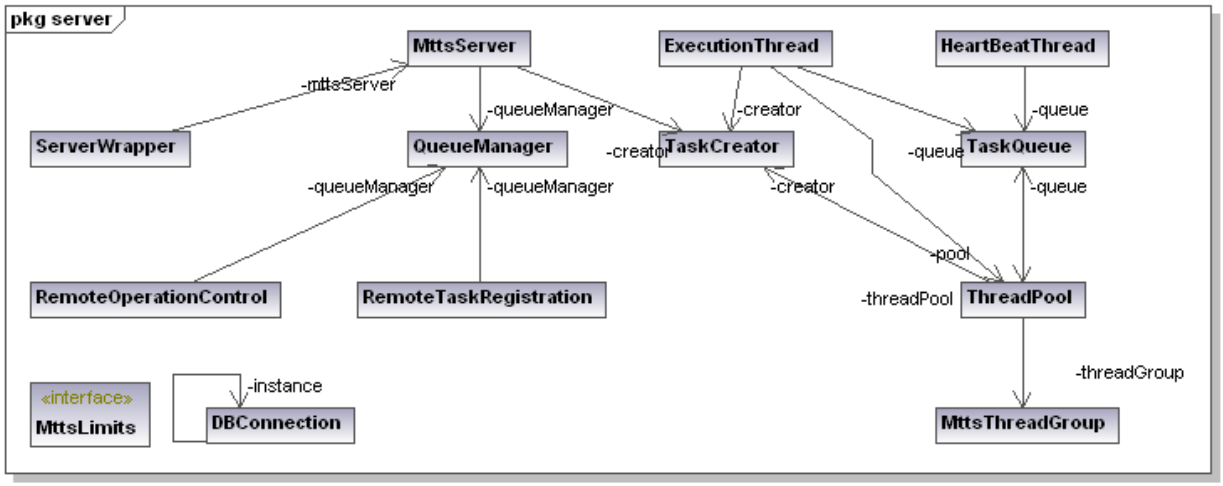


Figure 7: server Package Classes

7.1.2.1 server Package Classes:

This section provides the details of the associated classes in this package.

Class	Details
DBConnection	*This is a helper class to create and dispose connection with the database.*
MTTSLimits	*This class defines some constants (limits) such as max length of id , description, property value etc.*
MttsServer	*This is the main class and is responsible to start and stop the server.*
ServerWrapper	*This is wrapper class of the server and implements the functionality to run the server as a service.*
QueueManager	*This class creates queues by reading the information from configuration file. It maintains the record of created queues and its associated threads.*
TaskCreator	*This class is used to create a task and it loads the associated class (the class that is required to execute the task) into memory.*
ExecutionThread	*This class is a thread that executes the task. It uses the mutex algorithm (IMutex from IL package) to guarantee synchronized access to resources. The acquire and release functions of the mutex implementation are used to manage atomic block.*
TaskQueue	*This class stores tasks (created by TaskCreator) in a collection and then perform different operations over that collection, such as to list the frozen tasks, to list all the tasks etc.*
ThreadPool	*This class maintains (stores) the pool of created threads and is used to control the threads. The newly created threads are added to the pool, along with the facility to dispose of a thread.*
RemoteOperationControl	*This class remotely monitors the operations of the MTTS server. It provides functionality such as, list the tasks, list the tasks with errors, list the number of threads etc.*
RemoteTaskRegistration	*This class is used to add (register) task and its dependencies in the queue.*

7.2 il (Intelligent Lock) Package (from Incubator):

This package implements mutex algorithm that manages the synchronization among different

threads. The package provides a facility to acquire and release mutexes. This package has also classes that store the information about the status and statistics of mutexes. Here statistics means, how many mutextes has been created.

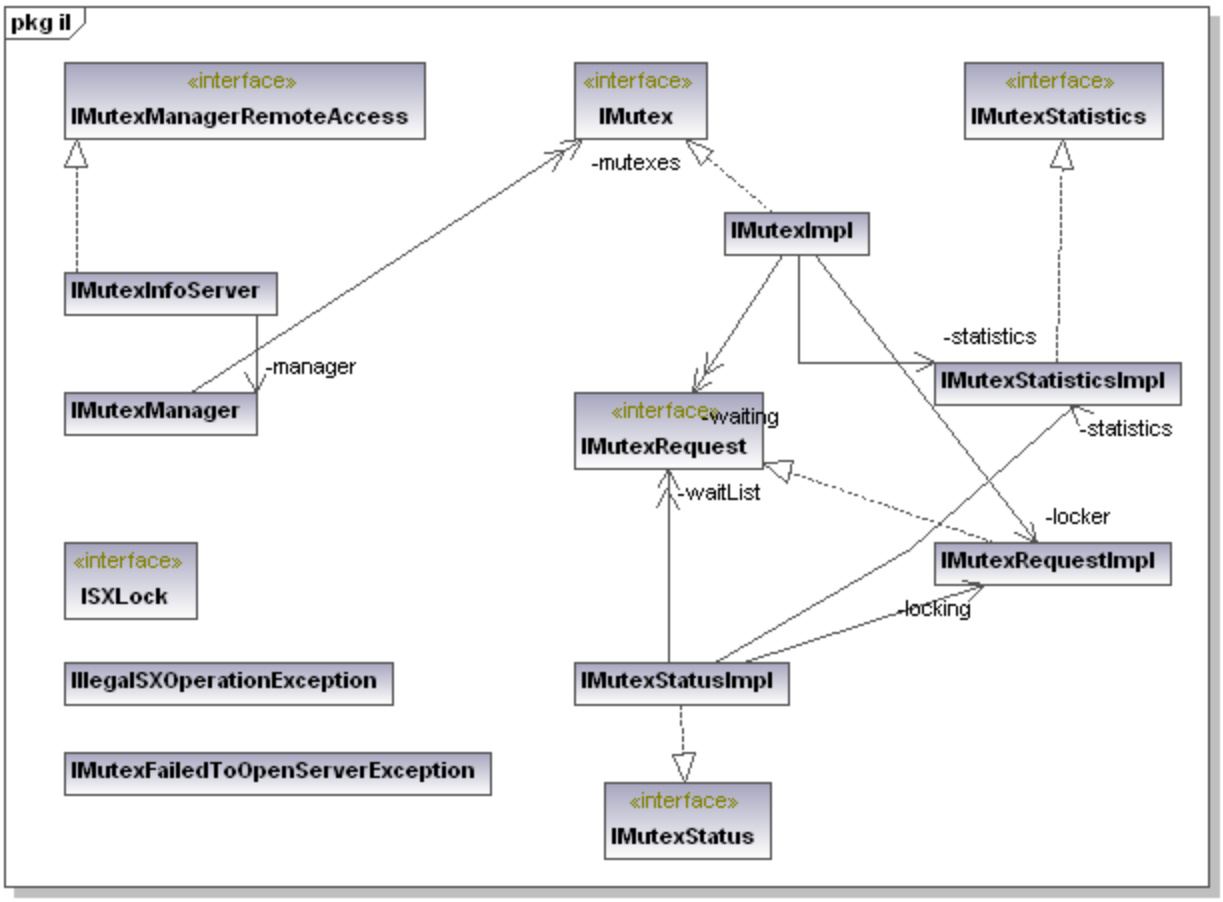


Figure 8: il Package Classes

7.2.1 il (Intelligent Lock) Package Classes:

This section provides the details of the associated classes in this package.

Class	Details
IMutex	*This interface represents a mutex that can be acquired by a single thread.*
IMutexImpl	*This class implements IMutex that can be acquired by a single thread.*

IMutexManagerRemoteAccess	*This interface provides information to the manager of mutexes to keep an eye over mutexes.*
IMutexRequest	*This interface represents a request to acquire a mutex.*
IMutexRequestImpl	*This class implements the IMutexRequest interface that is related with requests to acquire a lock.*
IMutexStatistics	*This interface represents the statistics of a lock such as average waiting time, acquire time and usage time.*
IMutexStatisticsImpl	*This class implements the interface IMutexStatistics which represents the statistics of a lock, such as average waiting time, acquire time and usage time.*
IMutexStatus	*This interface represents a snapshot of the lock state.*
IMutexStatusImpl	*This class implements interface IMutexStatus that represents a snapshot of the lock state.*
ISXLock	*This interface is about locks such as shared / exclusive locks, but this interface is not implemented.*
IllegalSXOperationException	*This is an exception class and the exception is thrown when there is a try to access a lock that does not exist.*

7.3 Model View Controller of MTTS:

In the following, we give a MVC (Model-View-Controller) architecture perspective of the MTTS application (see Figure below). *serverwrapper* and *MttsServer* classes are the controller classes whereas there is no view class. All the rest of classes belong to the model domain. The details and role of these classes is stated in the previous sections.

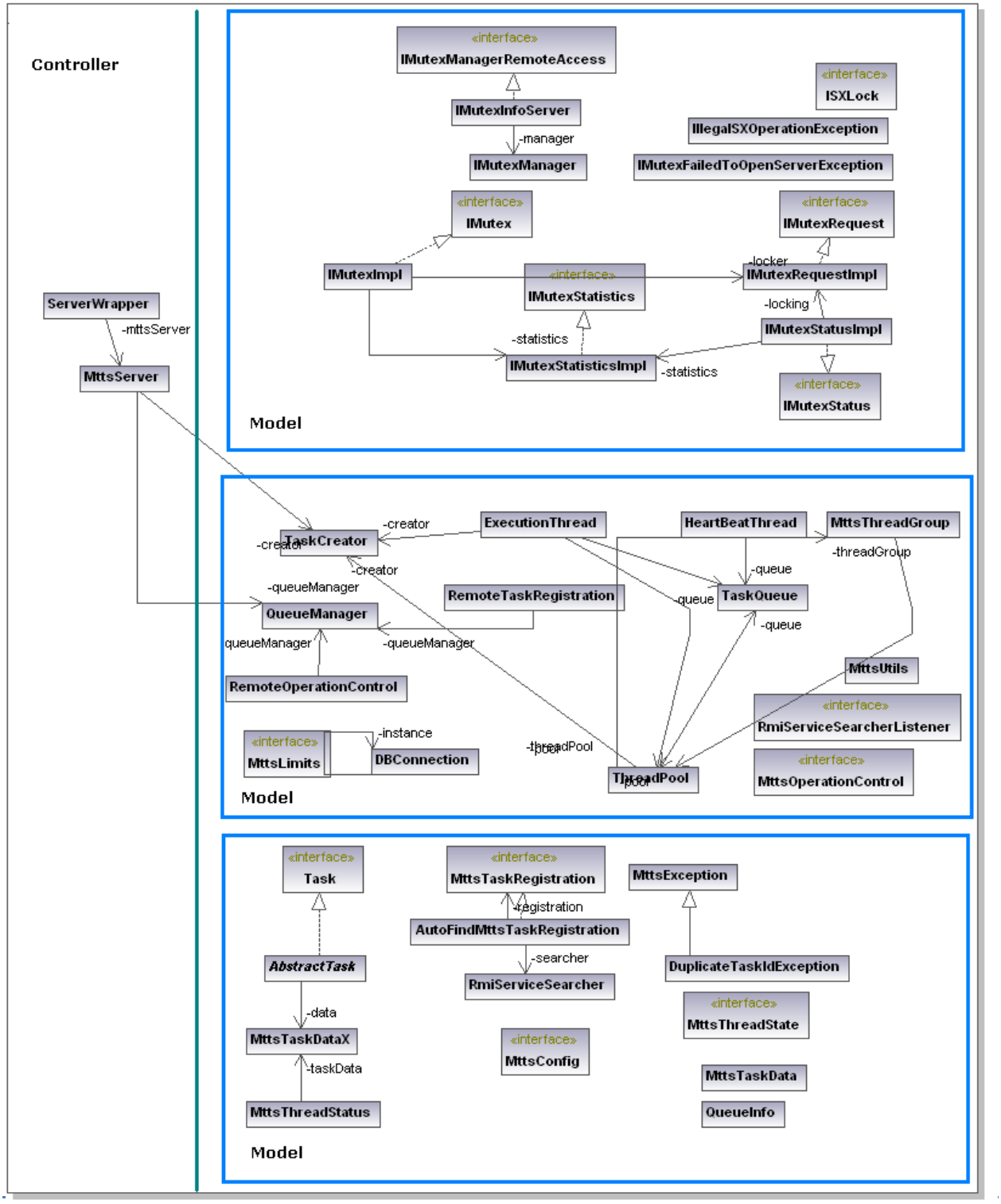


Figure 9: MVC of MTTS

8 Related Tools and Techniques:

This section describes the tools and techniques related with *Typestates* and *Access Permissions.* The former is used to specify and verify general aspects of program whereas later is used to specify and verify the concurrent behavior of program, indeed later is based on "Concurrency by default "paradigm.

8.1 TypeStates and Access Permissions:

The concept of *type* system is being used in programming language since the inception of early programming languages such as Pascal, Algol etc. The *type* system can be used to verify the program correctness (though only at a syntactic level) and history witnesses a significant debate among strictly typed languages and partially (loosely) typed languages with advantage and disadvantage of each. The type system provides information about the (not) possible operations performed over a variable and thus helps to write syntactically correct programs, but only syntactically correct.

The *Typestates* system is an extension of *type* system and provides a way of safe application of the operation over an object in a particular *Typestate*, thus unleash the semantic errors (not detected by the traditional *type* system). The *Typestates* system augments the existing strengths of *type* system, along with the information of possible *Typesates* changes (transitions) after the execution of operation. Consider the following snapshot of code for explanation.

```
Collection<Integer>  myBag;
Integer  item= myBag[5];
```

The traditional *type* system considers this code correct whereas *Typestates* system considers it incorrect. Traditional *type* system only knows (documents) that *myBag* is a collection of Integers and a read operation can perform over collection of items, whereas *Typestates* system knows (documents) more information about collection than mere its syntactic type and all possible operation on that syntactic type. In this particular context, *Typestates* system can also know (documents) that collection has further *Typestates*, *filled* and *empty* and a read operation

cannot performed for *empty* typestates and can be performed only for *filled* state. Obviously one has to specify these *Typestates* to get these benefits. So very briefly *Typestates* provides us a mechanism to verify the program correctness at a greater level than simple *type* system.

Typestates do not provide facility to reason about concurrency, however *Access permission* is a novel idea to express logical dependencies in the code about concurrent execution runs. *Access permissions* are abstract definitions on the capability of methods to access a particular state. Hence, a method should specify permissions to all the states it potentially accesses. Access permissions can be categorized into five types as described below. Additionally, Access permission uses data groups to express high-level specification of dependencies. Access permission is a more natural way to specify and verify the concurrent aspect of a program.

Unique: A unique permission to a reference guarantees that this reference is the only reference to the object at this moment in time. Therefore the owner has exclusive access to the object.

Immutable: An immutable permission to a reference provides non-modifying access to the referenced object. Additionally an immutable permission guarantees that all other existing references to the referenced object are also immutable

Shared: A shared permission to a reference provides modifying access to the corresponding object. Additionally a shared permission indicates that there are potentially other shared permissions (aliases) to the referenced object through which the referenced object can be modified.

Full: A full permission to a reference provides modifying access to the corresponding object. Additionally a full permission indicates that there are potentially other immutable permissions (aliases) to the referenced object through which the referenced object cannot modified.

Pure: A pure permission to a reference provides non-modifying access to the referenced object. Additionally a pure permission indicates that there are potentially other permissions (aliases) to the referenced object through which the referenced object can be modified.

8.2 Plural:

Plural is a sound modular *Typestate* checking tool for Java that employs fractional permissions to allow flexible aliasing control. Plural supports synchronized blocks and atomic blocks for checking concurrent programs. Plural's protocols can involve multiple interacting objects. The tool is an Eclipse plugin that relies on the Crystal static analysis framework. The main components of plural can be defined as

Crystal: Crystal is a framework for writing Java static analyses. It makes it easier to write dataflow analyses by creating three address codes out of Java code and by generating a control flow graph.

PlaidAnnotations: PlaidAnnotations is a very minor plugin and it contains the Java annotations to specifying a program, for example @Perm and @Share.

PluralAnnotationAnalysis: This plugin is used to check the well-formedness of specifications. For example, if there is a field in an invariant that does not exist, this analysis will produce a warning.

EffectChecker: This plugin is used to check that a particular method does have side effects.

FractionalAnalysis: This is main component of plural and is used to analyze the typestate for single-threaded programs.

NIMBYChecker: This plugin is used to do typestate analysis for multi-threaded programs by using the atomic blocks as means of mutual exclusion.

SyncOrSwim: This plugin is used for typestate analysis for multi-threaded programs that use Java's synchronized statement as their means of mutual exclusion.

8.3 Plaid:

Plaid is a radical new programming language designed for the nascent age of concurrency, and component-based computing. It focuses on "Concurrency by Default", a programming paradigm in which programmers specify dependencies between operations using permissions rather than specify the order of execution in the program. The run time system is therefore free to execute the program concurrently up to the expressed dependencies. It has a linear type system that tracks state changes dynamically using access permissions.

9 Work Plan:

This section provides the details of the work plan. The "*MTTSProcessingCode"* has three packages, "*mtts-api"*, "*server"*, "*il"* and around thirty five classes. The "server" package uses (calls) the "*mtts-api*" and "*il*" package. Keeping in view the mentioned sequence call, it is planned that we will specify the "*mtts-api*" package first, then "*il*" package and finally "*server*" package. It is further envisioned that as "*mtts-api*" package does not contain code regarding concurrency, so this package will be used for the verification of general properties whereas "*il*" and "*server*" package will be used to verify both general and concurrent properties due to its relevance with the concurrency.

9.1 Targeted Properties:

We divided the targeted properties into two main groups, "concurrent properties" and "general properties". The "concurrent properties" means the properties associated with deadlocks and race conditions whereas the "general properties" means, the properties associated with general aspect of program analysis such as non-null, reading from an empty collection, array out of index etc. Our focus is on "concurrent properties" and at abstract level we are interested to undertake a verification effort as described in the following items.

- Does the MTTS core use mutexes wisely? That is, is there any possibility that an improper usage of mutexes in the MTTS core leads to incorrect program behavior.

- If a thread acquires a lock and dies without releasing the lock. Now the threads waiting for that lock will never be able to acquire the lock, thus leading to the possibility of deadlock. Can we detect it statically by using our verification tools?
- Are the implemented mutexes are correct and do not leads to the problems of race conditions.

9.2 Time Plan:

This section provides the details of the time plan.

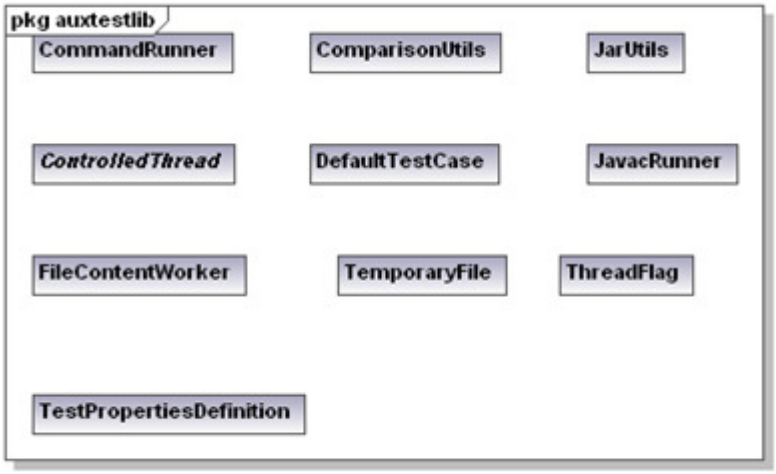
Activity	Time	Description
Plural/ Plaid Training	1-2 weeks	*This activity is related to learn Plural and Plaid tools.*
Specification of general properties	2 weeks	*This activity is related to specify the general properties.*
Specification of concurrent properties	4 week	*This activity is related to specify the concurrent properties.*
Compilation of results	2 week	*This activity is related with report and paper writing.*

10 APENDIX:

The code (that Novabase provided) can be divided into 3 subgroups, namely "*ProcessingCode*"," *GUICode"* and *"TestcaseCode"*. The "*TestcaseCode*" is related with test cases and "*GUICode*" is related with GUI. The "*ProcessingCode"* can be further subdivided into two main groups, "*MTTS-ProcessingCode"* and *"nonMTTS-ProcessingCode"*. This section provides the details of "*TestcaseCode*"," *GUICode and "nonMTTS-ProcessingCode"*. **Notice that, this code is not related with MTTS working, we are providing the details just for the purpose of information.**

10.1 auxtestlib Package:

This package is not used in the main source code rather it is used in the code of test cases of other packages. It provides the facilities to write test cases. It also provides facilities such as command line operations, reading and writing from files, manipulating directories and jar files etc.



10.1.1 auxtestlib Package Classes:

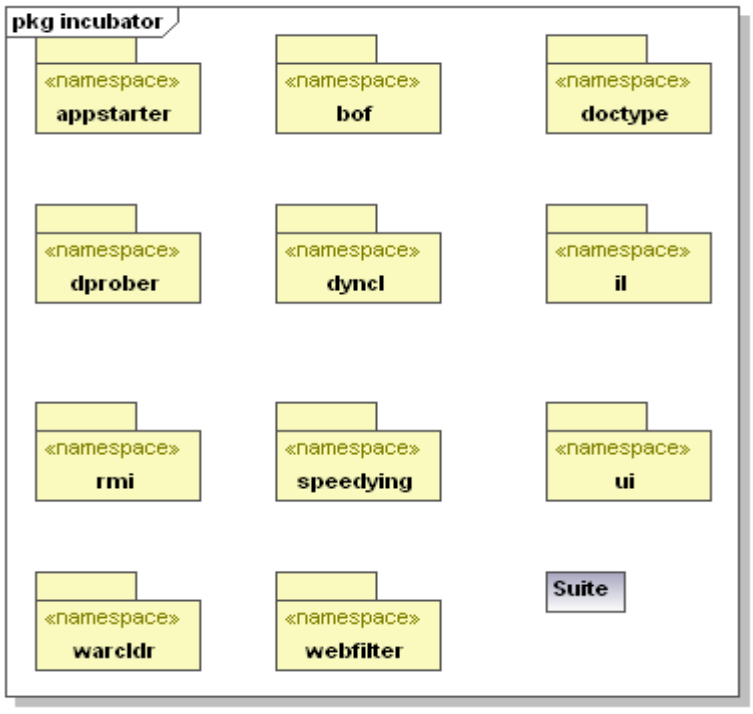
This section provides the details of the associated classes in this package.

Class	Details
CommandRunner	*This Class is used to run a command and capture the output*

ComparisonUtilis	*This class is used to Verify that two arrays are equal. Two arrays are equal if they have Same length and all its objects are equal*
JarUtils	*This Class provides utility methods for working with jars, such as to load jar files etc.*
DefaultTestCase	*This class provides methods to access the properties of the tests and provide basics services for all tests.*
FileContentWorker	*This class provides utilities to simplify working with the contents of text files.*
TemporaryFile	*This class is used to create (delete) a temporary file (directory)*
ThreadFlag	*This class is used to control the running threads.*
TestPropertiesDefination	*This class defines properties for testing.*

10.2 Incubator Package:

This package is a general purpose library that is being used (can be used) by a number of projects. MTTS uses some of its packages; the most notable is il (Intelligent Lock). Most of the packages contain a GUI part enclosed inside the main package to perform the GUI related tasks.



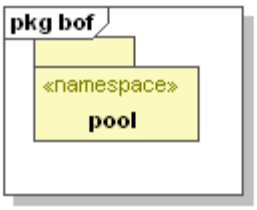
10.2.1 appstarter Package:

This package contains the interface to run two of the modules belong to the package incubator, namely "*Intelligent Lock Monitor*" and "*Pool Manager*"

ClassName	Purpose
`Starter`	*This Class provides a GUI to run two modules "Intelligent Lock Monitor" and "Pool Manager".*

10.2.2 bof package:

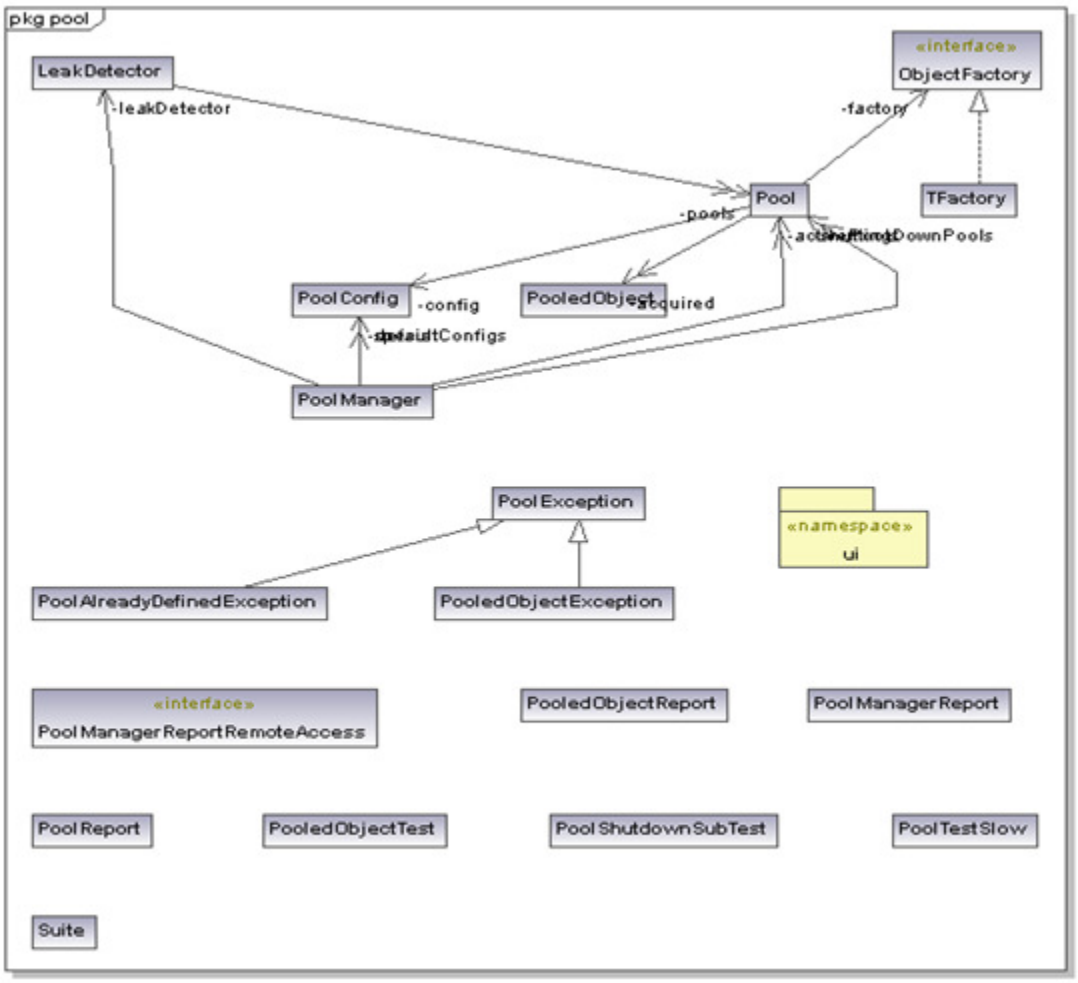
This package contains another package pool.



10.2.3 pool package:

This package is used to create business objects (used by some other application of Novabse) and then make a pool of the created objects. The package also provides facility to manage (monitor) pooled objects. The package provides facility to handle leakage of objects. We think

that might be leakage of objects is related with aliasing and explored this package and concluded that leakage of object is not related with aliasing; rather the leakage concept is used for some other purpose.



10.2.3.1 pool Package Classes:

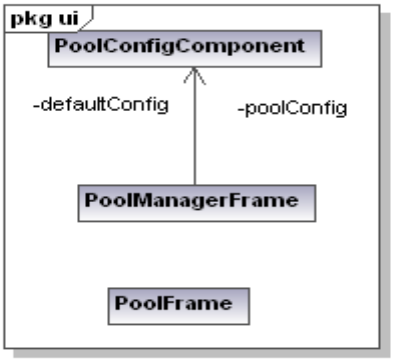
This section provides the details of the associated classes in this package.

Class	Details
`Leakdetector`	*This class performs the detection of leaks of objects.*
`ObjectFactory`	*This interface is a Factory that creates and destroys*

	objects.
`Pool`	*This class manages a pool of objects.*
`PoolConfig`	*This Class defines the configuration of a pool.*
`PoolObject`	*This Class implements an object that can be pooled for use in a Thread. Each object may or may not be acquired by a thread.*
`PoolException`	*This class is used as super class of other exceptions .*
`PooledObjectException`	*This Exception is thrown when it fails to acquire or release an object*
`PoolAlreadyDefinedException`	*This Exception is thrown when creating an already existing pool.*
`PoolManagerReportRemoteAccess`	*The Interface for remote access to the pool manager's report.*
`PooledObjectReport`	*This class represents the information of acquisition of an object from the pool.*
`PoolReport`	*This class defines the current status report of a pool.*

10.2.4 ui Package:

This package is related with user interface.



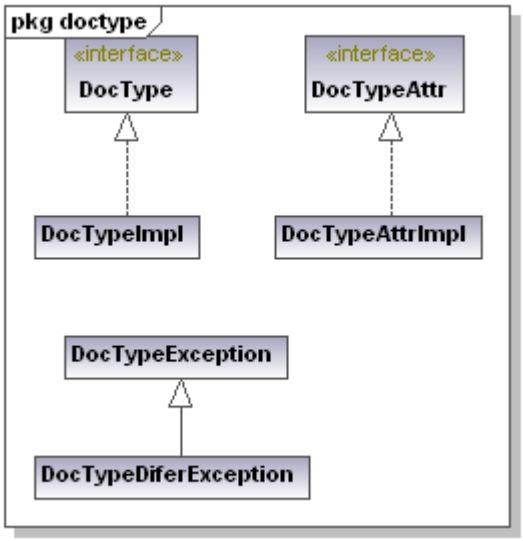
10.2.4.1 ui Package Classes:

This section provides the details of the associated classes in this package.

Class	Details
PoolConfigComponent	*This UI class provides different textboxes to store pool configuration. It may be considered as an input dialog.*
PoolManagerFrame	*This UI class is used to display data from PoolManager. It may be considered as an output dialog(window)*
PoolFrame	*This UI class displays the main window "Pool Monitor"*

10.2.5 doctype Package:

This package defines document and its attributes along with the associated exceptions. The document has properties like size, type and name etc. This information can be associated with tasks for further manipulation. This package is documentum specific. documentum is a property software used by the business community for document management system.



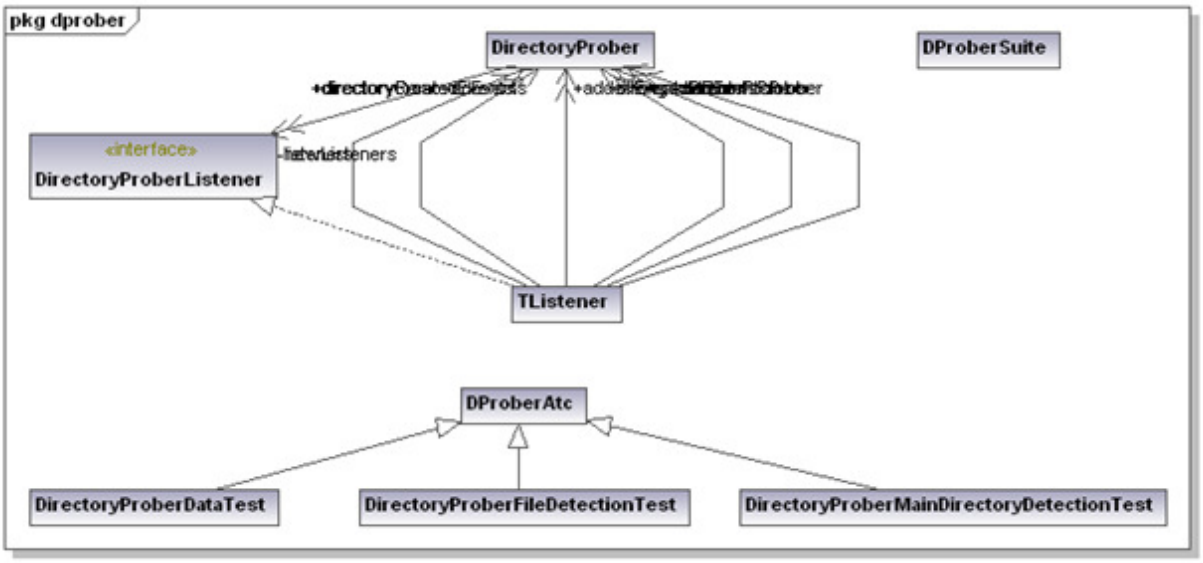
10.2.5.1 doctype Package Classes:

This section provides the details of the associated classes in this package.

Class	Details
DocType	*This interface represents a kind documentary.*
DocTypeImpl	*This class implements a documentary.*
DocTypeAttr	*This interface represents attributes of a documentary type.*
DocTypeAttrImpl	*This class represents attributes of a documentary type.*
DocTypeException	*This class is a superclass for all exceptions in package doctype.*
DocTypeDiferException	*This exception indicates that the type of a documentary is different that expected.*

10.2.6 dProber Package:

This package defines operations that observe the changes in a directory and inform the listener accordingly.



10.2.6.1 dProber Classes:

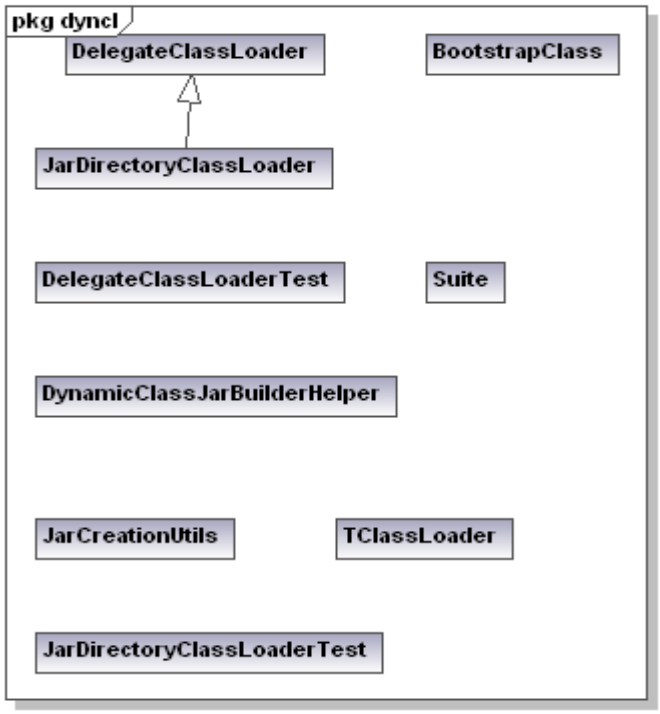
This section provides the details of the associated classes in this package.

Class	Details
DirectoryProber	*This class informs the listeners about the directory changes such as (creation, deletion or modification of files).*

DirectoryProberListener	*This is a Listener that get information when directory is being changed.*
DProberAttc, DirectoryProberDataTest, DirectoryProberFileDetectionTest, DirectoryProberMainDirectoryDetectionTest, DProberSuite	*These classes are related with testing of this package.*

10.2.7 dyncl Package:

This package is used to dynamically load classes and jar files. This package can be considered as an auxiliary package that has no role in the business logic of application; however it may be used to create services. Here services means to run application in a window (linux) service mode.



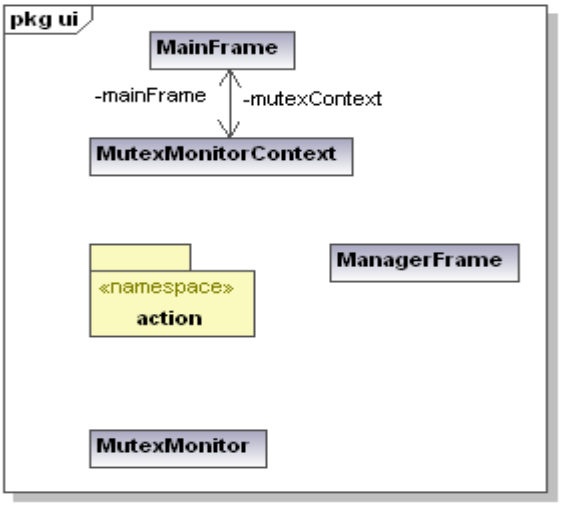
10.2.7.1 dyncl Package Classes:

This section provides the details of the associated classes in this package.

ClassName	Purpose
`DelegateClassLoader`	*This is an interface to load files (jars typically) from a directory. The directory can be read recursively or not. Here recursively means, to read from the inner directories.*
`JarDirectoryClassLoader`	*This class loads all files (jars typically) from a Directory. The directory can be read recursively or not.*
`BootstrapClass, DelegateClassLoaderTest, DynamicClassJarBuilderHelper, JarCreationUtils, JarDirectoryClassLoaderTest, Suite, TClassLoader`	*These classes are related with testing of this package.*

10.2.8 Ui Package:

This package is related with the user interface to check the status of each mutex.



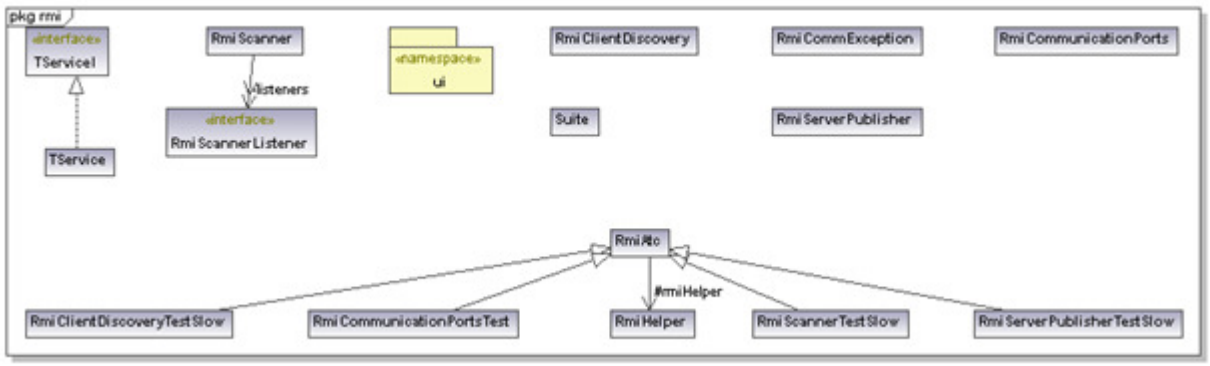
10.2.8.1 Ui Package Classes:

This section provides the details of the associated classes in this package.

Class	Details
`MainFrame`	*The class displays the main application window for monitoring mutexes. This window serves as a container for the various elements, such as display boxes.*
`MutexMonitorContext`	*This class holds the context of monitoring implementation of mutexes*
`ManagerFrame`	*It provides the main function to call MainFrame.*
`MutexMonitor`	*This class also provides the main function to call MainFrame.*

10.2.9 rmi Package:

This package is used for publishing and finding RMI services.



10.2.9.1 rmi Package Classes:

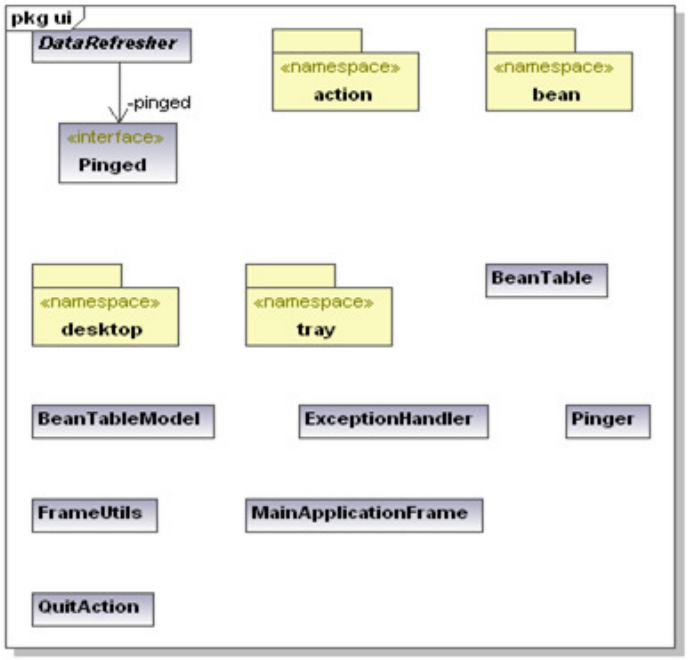
This section provides the details of the associated classes in this package.

Class	Details
`RmiClientDiscovery`	*This class is responsible to find out RMI clients.*
`RmiCommunicationPorts`	*This class defines the default range of ports for communication and obtains the maximum and minimum values for ports.*

RmiScanner	*This class is used to scan for a server.*
RmiServerPublisher	*This class is used to publish the RMI interfaces.*
RmiCommException	*This exception is thrown by a layer of communication via RMi.*
UI	*The UI Package is related with UserInterface.*

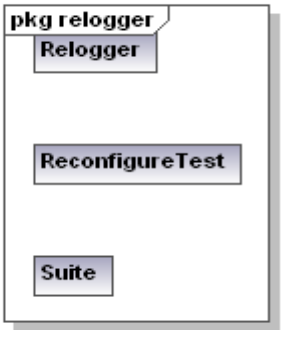
10.2.10 Ui Package:

This package is related with user interface:



10.3 relogger Package:

This Package is used for configuration.



10.3.1 relogger Package Classes:

This section provides the details of the associated classes in this package.

Class	Details
`relogger`	*This Class contains methods to initialize the polling of the fileConfiguration.*

10.4 Model View Controller Architecture of nonMttsCode:

This section shows the model view controller architecture of the *nonMttsCode* . The UI packages are performing the job of controller and view both.