UDK 004.9:504:519.6

DISTRIBUTED ARCHITECTURE OF RODOS - DECISION SUPPORT SYSTEM FOR NUCLEAR EMERGENCY MANAGEMENT IN EUROPE

Ie. Ievdin, D. Treebushny, M.I. Zheleznyak
Institute of Mathematical Machine and System Problems NAS of Ukraine
e-mail: yewgen@env.com.ua

INTRODUCTION

RODOS system for nuclear emergency management in Europe has been developed under the funding of European Commission since 1992 (Raskob W., 2007). RODOS has been designed as a complex system which includes models and data bases used for modeling, evaluation of possible nuclear emergency consequences and for the corresponding emergency planning. In 2006 the work on the RODOS system redesign using modern Java and GIS technologies was started (Ievdin Ie., et al, 2007; 2008).

Redesigned RODOS system – JRodos – is developed as a distributed system with the Computational Engine entity for performing calculations, the Management Server entity for database access, collecting information from other parts of system and the Client entity for capturing input information, visualizing results etc. Important step in the distributed architecture implementation is physical separation of computational processes and the main system process to improve performance and stability of the latter. Open-source library Cajo is used as a middleware between Computational Engine and calculation processes. This allows concentrating on system business logic rather than communication details.

CAJO AND MULTICAST FUNCTIONALITY

The cajo project is a small, 100% pure Java, open source library, enabling powerful dynamic multi-machine cooperation; both within and between, both free and proprietary Java applications. It provides an easy to use, yet completely understandable framework which allows multiple remote Java Virtual Machines (JVM) to work together seamlessly, as one.

Cajo library provides easy access to multicast functionality. First, it allows an object to "broadcast" a remote reference to itself over the network. Second, it allows an object to listen for and catch these reference broadcasts. Once caught, an object can invoke the public methods of the broadcasting object. Multicast allows objects to discover and access each other spontaneously, without the need for static registries.

There are two parameters on which the objects must agree in order to send and receive references between each other. The pair {IP address, port number} defines a specific "channel", to use a radio analogy. Changing either of these values effectively switches to an entirely different channel. This UDP/IP address/port pair is fundamentally different from a TCP/IP address in that it doesn't identify a specific machine and socket; rather it represents all machines that are members of a "group". The cajo multicast concept is illustrated in the following diagram:

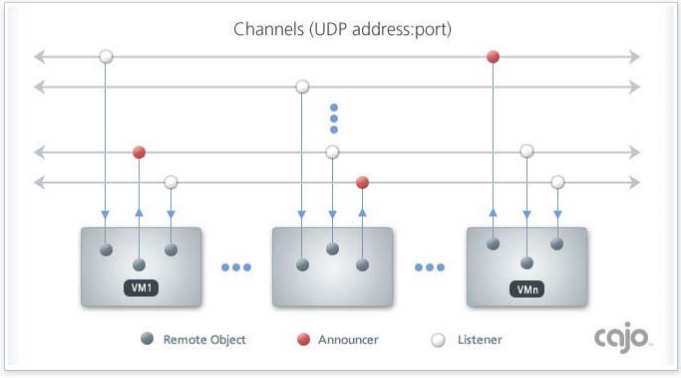

Figure 1. Multicast concept diagram.

IMPLEMENTATION IN THE REDESIGNED RODOS

Described functionality is used in the redesigned RODOS for running computational models as separate OS processes independent from the main system process. Computational models are native dynamic-link libraries (dll) which are loaded in runtime for numerical calculation. This approach protects main RODOS system against bugs in models which can be memory leaks, uncaught exceptions, critical errors and unpredictable termination of the process. At the same time running models in the separate Java Virtual Machine (JVM), not as standalone executable, allows to preserve all advantages of dataitems concept and model wrapper approach (Donchyts G., et al. 2003; Ievdin Ie., et al, 2007). This includes to name just a few: using common interface and data structure for initializing, running models and receiving results; exchanging data via memory and avoiding unreliable time cost files; establishing call-backs for interaction with complex models and parallel run of the models.

The following timing diagram describes the implementation of multicast functionality in the JRodos system. Communication between the main JVM and its daughter JVM (Number Cruncher JVM used for numerical calculation) is done via a multicast channel. This channel is defined by the special UDP address dedicated to Cajo and a port selected to be unique for each of the models calculating in parallel.

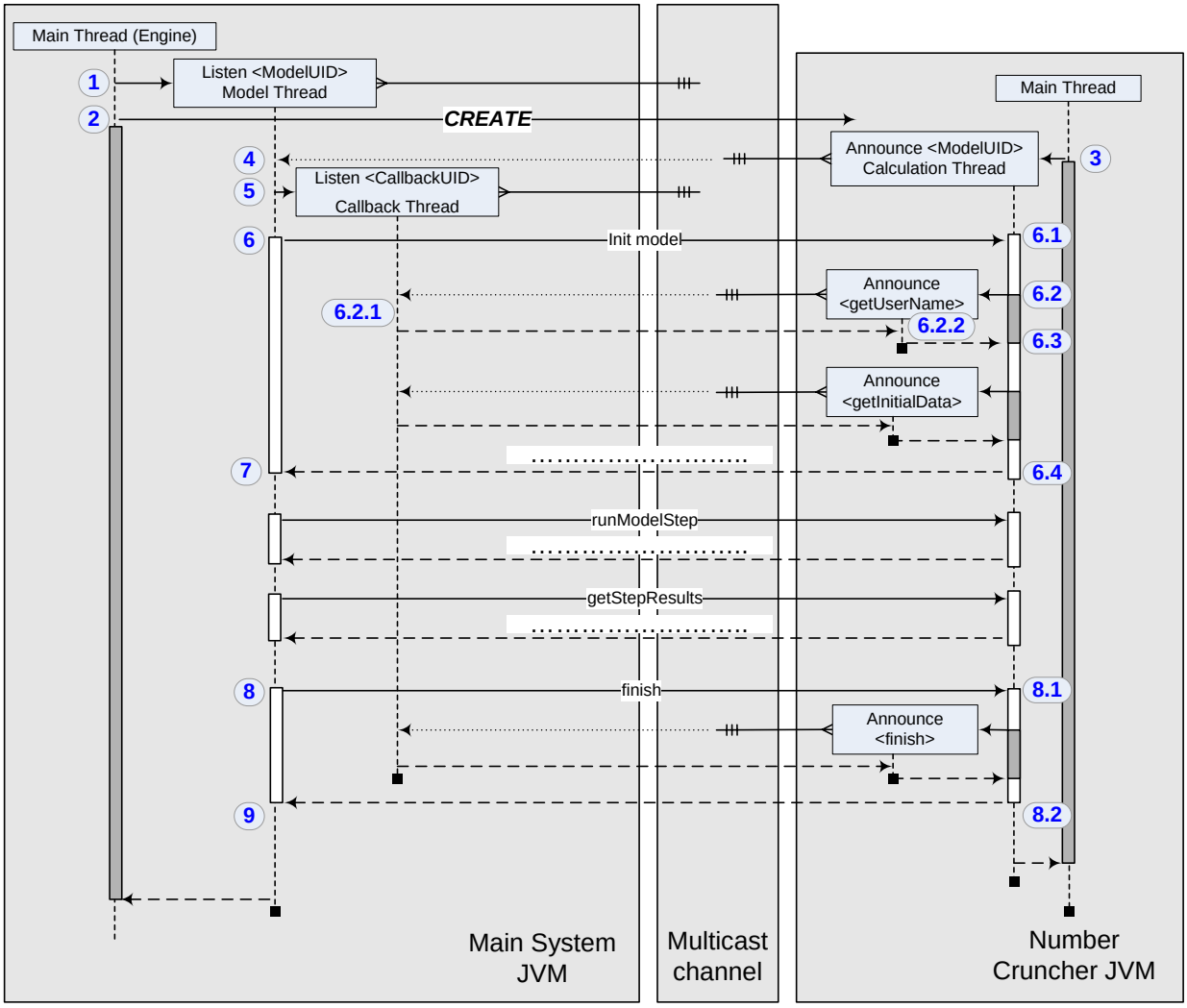

Figure 2. Implemented multicast functionality timing diagram

Explanation of the diagram:

1. The *Main Thread (Engine)* starts listening the specified IP and port, reacting only on the announcements containing the specified globally unique ID (ModelUID). This call creates a thread (Model Thread) which keeps on listening until the successful announcement.
2. The *Main Thread (Engine)* runs external command to create a new JVM holding the information needed to a) connect to the Main System JVM and b) run the desired type of model. The parameters are: the already defined ModelUID, IP, port and name for model class (ModelEntryPointClass). After creation the Main Thread (Engine) waits for notification from other threads.
3. The *Main Thread of the Number Cruncher JVM (NC JVM)* makes an announcement with the specified ModelUID. This call creates a new *Calculation Thread* in which all the calculation tasks will be performed. The *Main Thread of the NC JVM* waits for notifications.
4. The *Main Thread (Engine)* receives the announcement and recognizes its own ModelUID. Now the two JVMs are connected, so it is possible to perform Remote Invoke commands. The top level model logic is ready to be executed.
5. The *Main Thread (Engine)* creates a new *Callback Thread* listening for the specified CallbackUID.
6. The block "Init Model" starts. The remote *initModel(...)* method is called on the *NC JVM*.

   6.1. *initModel(...)* logic starts.

   6.2. It usually happens that during the execution of the *initModel(...)* logic some additional parameters are needed, which available only on the Main System JVM. Thus the callback mechanism is introduced. *Calculation Thread* announces for the remote method *<getUserName>* and provides CallbackUID to be able connecting to *Callback Thread* of the *Main System JVM*. This creates an additional "short living" thread waiting for the reply. *Calculation Thread* waits.

   6.2.1. *Callback Thread* of the *Main System JVM* receives the remote call, processes it and sends back the result.

   6.2.2. "Short living" thread receives the result, notifies the *Calculation Thread* and normally finishes.

   6.3. *Calculation Thread* wakes up and proceeds executing its logic. During the execution other callback may be needed. These are processed using the described steps 6.2 – 6.2.2.

   6.4. *initModel(...)* logic ends, giving the control back to *Model Thread* of the *Main System JVM*.

7. *Model Thread* of the *Main System JVM* receives the control and process further methods of the top level model logic.

   Usually the steps are *runModelStep(...)*, *getStepResults(...)* etc. The calls are processed as it is described in the items 6 – 7.

8. Final call *finish()* is performed from the *Model Thread* of the *Main System JVM*.

   8.1. *Calculation Thread* of the *NC JVM* receives this call and send the *finish()* command to the *Calback Thread* via "short living" thread. *Callback Thread* sends the respond to the "short living" thread and finishes its lifetime.

8.2. *Calculation Thread* receives a control, gives the reply to the *Model Thread* of the *Main System JVM*, wakes the *Main Thread* of the *NC JVM*. Both *Calculation Thread* and the *Main Thread* of the *NC JVM* finish their lifetime.

9. M*odel Thread* gives the control back to the Main Thread (Engine) and finishes its lifetime.

CONCLUSIONS

Implemented functionality is an important step in the development of the redesigned RODOS as a distributed system. Separation of a computation model process and the main system process significantly increases stability and performance of the whole application. Computational models work with their own memory pool which is independent on the memory allocated to the system. Any instabilities, crashes, memory leaks have no influence on the main RODOS system. With this approach JRodos system is adopted to be run on multicore / multiprocessor machines and is ready for calculating several computational models in parallel.

At the same time running each model in a separate Java Virtual Machine (JVM), not as standalone executable, allows using advantages of dataitems and model wrapper concepts. To make this possible the dataitem classes and JRodos core functionality are extracted from the system into libraries which are used both in the main system and model plug-ins. Plug-ins are searched and loaded if needed in runtime. All model specific functionality (user interface, model wrapper, data structure) is taken into a corresponding plug-in. This allows to perform integration of new models independently on the main system.

The same Multicast approach will be used for further implementation of the distributed system, separation of the Computational Engine, Management Server and Client entities.

The system with the implementation of the described functionality has been released within the context of JRodos Demonstration project which involves users from different organization to test the redesigned RODOS system on their national conditions. Issues and requests are collected through the established Bugzilla engine which allows conducting JRodos development according to the real needs of the end-users.

REFERENCES

1. Raskob W. European approach to nuclear and radiological emergency management and rehabilitation strategies (EURANOS). Kerntechnik 72 (4) (2007) 172-175.

2. Ievdin Ie, Prymachenko O., Shlyakhtun N., Kovalets I., Treebushny D., Donchyts G., Chorny I., Guziy O., Zheleznyak M.I., First prototype of redesign of the RODOS system for nuclear emergency management in Europe, Зб. праць Третьої конференції з міжнародною участю "Системи підтримки прийняття рішень: теорія і практика. СППР-2007", 7-27 Червня 2007 р., Київ, С. 22-25.

3. Ievdin Ie, Treebushny D., Kovalets I., Guziy O., Zheleznyak M.I., Development multi-platform version of decision support system for nuclear emergency management in Europe (RODOS) on base of modern Java and GIS technologies, Зб. праць Четвертої конференції з міжнародною участю "Системи підтримки прийняття рішень: теорія і практика. СППР-2008", червень 2008 р., Київ, С. 121-124.

4. Cajo project, https://cajo.dev.java.net/

5. Donchyts G., Zheleznyak M.I., 2003. Object-Oriented Framework for Modelling of Pollutant Transport in River Network. In Computational Science – ICCS 2003, Lecture Notes in Computer Science, Springer, Vol. 2657/2003, 0302-9743.