Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Jan Čurn

Distribution for Open Modelling Interface and Environment

Katedra softwarového inženýrství

Vedoucí diplomové práce: *Doc. Ing. Petr Tůma, Dr.*

Studijní program: *Informatika, Softwarové systémy, Architektura a principy systémového prostředí*

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 10.8.2007

Jan Čurn

Table of Contents

List of Figures

List of Tables

Abstract

Název práce: *Distribuce pro Open Modelling Interface and Environment*
Autor: *Jan Čurn*
Katedra: *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *Doc. Ing. Petr Tůma, Dr.*
E-mail vedoucího: *petr.tuma@mff.cuni.cz*
Abstrakt:

OpenMI je standard pro propojování simulačních modelů vody a životního prostředí. Standard a v současnosti dostupný podpůrný software ovšem podporuje pouze simulace běžící na jednom počítači, v jednom vlákně.

Cílem práce bylo vytvoření systému schopného propojovat OpenMI modely běžící na různých počítačích pomocí síťového podsystému. Systém se skládá z uzlových serverů, které poskytují přístup k modelům registrovaným klienty. Klienti zpřístupňují své lokální modely serverům a také umožňují stávajícímu OpenMI softwaru transparentně přistupovat ke vzdáleným modelům registrovaným jinými uživateli.
Klíčová slova: *OpenMI, distribuovaný systém, integrující server*

Title: *Distribution for Open Modelling Interface and Environment*
Author: *Jan Čurn*
Department: *Department of Software Engineering*
Supervisor: *Doc. Ing. Petr Tůma, Dr.*
Supervisor's e-mail address: *petr.tuma@mff.cuni.cz*
Abstract:

OpenMI is a standard used to link water and environmental models. However, the standard and the currently available supporting software only support single-computer single-threaded simulations.

The thesis delivers a system capable of linking OpenMI models across computers using their network subsystem. The system consists of hub servers that provide access to models registered by clients. The clients make local models accessible to the servers and also provide the legacy OpenMI software with a transparent access to remote models registered by other clients.
Keywords: *OpenMI, distributed system, integrating server model*

1. Overview

OpenMI is European standard (see [Omi05]) for linkage of computational models in domain of water and environment. As defined by the OpenMI Standard, the models are independent software components (objects), accessible using well-known interfaces. OpenMI also provides set of tools enabling users to link the models and run the simulations on them. Currently, OpenMI is designed for a single-computer environment. The models may internally utilize any remote resources on different computers; however, there is no standardized way how to link OpenMI models running on different computers.

The goal of this thesis is to deliver a system, which will enable users to link any OpenMI models running on different computers. The system consists of hub servers and clients. The servers are used to intermediate communication between clients, and to register the models provided by clients. The clients make local models accessible to the servers and also provide the legacy OpenMI software with a transparent access to remote models registered by other clients.

The OpenMI simulation uses pull-driven mechanism, which means that one model invokes methods of another inter-linked model, which may then invoke methods of another model etc. As defined by the OpenMI Standard, all calls between models are synchronous, i.e. run in single thread of execution. Several models depend on this aspect and behave incorrectly if their methods are called from different threads.

In fact, our task is to distribute the single-thread call-stack to several computers, whilst using the integrating server model. The OpenMI Standard has not been designed to simplify remote access to models, so there are several issues which prevent simple adoption of common remoting techniques. Next challenge is to integrate or implement right communication protocol, since our distributed system poses quite complex requirements on it. Considerable task is also to optimize the overall performance of the system.

The outline of the text is following: Section 2 gives a brief introduction to such parts of the OpenMI standard that are necessary to understand the further text. Section 3 describes the basic ideas behind our distributed system and discusses its use cases. Section 4 then in detail describes the components of the system and how they interact with each other. Next, Section 5 introduces the concept of integrating server and discusses how it can be incorporated in our system. Section 6 discusses what communication protocol has been used and why. In Section 7 we explain the performance optimizations which have been necessary to speed-up the distributed OpenMI computation. Section 8 briefly describes the particular software components of our implementation. Last sections of the thesis conclude the work done (Section 9), suggest the possible future work (Section 10) and list the used literature (Section 11).

2. Introduction to OpenMI

This section gives a brief introduction to the OpenMI Standard (see [Omi05]), which is necessary to understand the further text. The readers already familiar with the standard may skip this section.

OpenMI is a shortcut for *Open Modelling Interface and Environment*, a standard for model linkage in the domain of water and environment. The *OpenMI Standard* defines the set of interfaces, which enable computational models to interact with each other. The standard is very generic allowing the linkage of different kinds of models from different disciplines like atmosphere processes, rainfall-runoff, river hydraulics, flooding, sewerage, water distribution, fishing …etc. The OpenMI interfaces are not bound to a particular platform, generally, component written in any language, running on any platform, may be *OpenMI compliant*, if it fulfils the requirements defined by the standard. However, the primary platform for which the implementation of OpenMI interfaces is available, is Microsoft .NET Framework (currently, OpenMI version 1.2.0 is targeted for .NET 2.0). The whole description of the OpenMI Standard may be found in [Omi05].

Besides the interfaces, OpenMI provides a user interface application enabling the users to link models and run the simulations (*deployment software*) and also delivers supporting libraries which simplify the process of both development of new models and migration of the legacy models to OpenMI, without the need of rewriting the cores of the computational engines. This software is available only for Microsoft .NET Framework, as well.

2.1. Linkable Component

Linkable component (LC) is elementary part of OpenMI. It represents a single computational model, i.e. computational engine populated with the data. The access to linkable component is abstracted using *ILinkableComponent* interface. Additionally, the linkable component may implement *IDiscreteTimes* interface (to inform the callers that it computes values in discrete time steps) and/or *IManageState* interface (to persist the state of the computation). Table 1, Table 2 and Table 3 gives a brief description of all methods of these interfaces:

Method/property	**Description**
Initialize	Initializes the LC using array of arguments (*IArgument*).
ComponentID	Gets the string identifying the computation engine.
ComponentDescription	Gets the string with description of the computation engine.
ModelID	Gets the string identifying the model (i.e. computation engine + data).
ModelDescription	Gets the string with description of the model.
InputExchangeItemCount	Gets number of input exchange items.
GetInputExchangeItem	Gets the n-th input exchange item (*IInputExchnageItem*).
OutputExchangeItemCount	Gets number of output exchange items.
GetOutputExchangeItem	Gets the n-th output exchange item (*IOutputExchangeItem*).
TimeHorizon	Gets simulation time horizon of this model (*ITimeSpan*).
AddLink	Adds a single link (*ILink*) between this LC and other LC.
RemoveLink	Removes specific link identified by its ID string.
Validate	Checks whether the computation may start on this model, i.e. that *Prepare* method can be called. If an error is encountered, the method returns a string with description of that error.
Prepare	Prepares the computation.
GetValues	Gets the value for specific time (*ITime*) and output link (identified by ID string). The call typically invokes the computation in the engine. The *GetValues* method returns an instance of *IValueSet* interface.
EarliestInputTime	Gets the earliest time (*ITimeStamp*), for which this LC needs input from other inter-linked LCs. These LCs may use this property to clean their internal buffers.
Finish	Finishes the computation. The models typically write their result files to disk when this method is called.
Dispose	Releases all resources associated with the LC.
GetPublishedEventTypeCount	Gets the number of events published by the LC.
GetPublishedEventType	Gets the n-th published event type (*EventType*).
Subscribe	Subscribes an event listener (*IListener*) to specific event type (*EventType*).
UnSubscribe	Unsubscribes the event listener (*IListener*) to specific event type (*EventType*).
SendEvent	Sends the event (*IEvent*) to listeners subscribed for corresponding event type.

Table 1 Methods and properties of *ILinkableComponent* interface

Method	**Description**
HasDiscreteTimes	Gets boolean value indicating that values of a specific combination of quantity (*IQuantity*) and element set (*IElementSet*) are defined on discrete time steps.
GetDiscreteTimesCount	Gets the number of discrete time steps for specific combination of

	quantity (*IQuantity*) and element set (*IElementSet*).
GetDiscreteTime	Gets the n-th discrete time step for specific combination of quantity (*IQuantity*) and element set (*IElementSet*).

Table 2 Methods of *IDiscreteTimes* interface

Method	**Description**
KeepCurrentState	Saves the current state of LC and returns the ID string for that state.
RestoreState	Restores the state identified by ID string.
ClearState	Removes the state identified by ID string from the internal storage.

Table 3 Methods of *IManageState* interface

2.2. OMI File

In OpenMI, the model is described using *OMI* file. It is a XML file containing the information which type implements the *ILinkableComponent* interface and in which .NET assembly it resides. Additionally, OMI file contains arguments which must be supplied to *Initialize* method to initialize the LC properly (e.g. a list of simulation input files). Deployment software uses the information from OMI file to locate the assembly, load it into the memory, instantiate the linkable component and initialize it using supplied arguments.

2.3. Model Linkage

During the computation the inter-linked OpenMI models exchange the data with each other. Exchange items define where and what data may be exchanged. More precisely, single exchange item is a combination of *element Set* (*IElementSet*) saying "where to exchange" and *quantity* (*IQuantity*) saying "what data to exchange". For example, in river hydraulics model, the element set may be a river cross-section and the quantity may be a water discharge. The link from model A to model B is a combination of one A's output exchange item and one B's input exchange item. The set of inter-linked LCs is referred to as *OpenMI composition*.

An input exchange item is abstracted using *IInputExchangeItem* interface, an output exchange item using *IOutputExchangeItem* interface. Table 4 and Table 5 show properties and methods of these interfaces.

Method	**Description**
Quantity	Gets the information about what data will be exchanged (*IQuantity*).

ElementSet	Gets the information about where the data will be exchanged (*IElementSet*).

Table 4 Methods of *IInputExchangeItem* interface

Method	**Description**
Quantity	Gets the information about what data will be exchanged (*IQuantity*).
ElementSet	Gets the information about where the data will be exchanged (*IElementSet*).
DataOperationCount	Gets the number of data operations associated with this output exchange item.
GetDataOperation	Gets the n-th data operation (*IDataOperation*).

Table 5 Methods of *IOutputExchangeItem* interface

The data operations defined by output exchange items are used to transform the data produced by the model (e.g. linear transformation, spatial interpolation…). The list of methods and properties of *IQuantity*, *IElementSet* and *IDataOperation* interfaces may be found in the [Omi05]. For our purposes their further explanation is unnecessary.

To link two models, the *AddLink* method must be called on both source and target LC, supplying an implementation of *ILink* interface as argument. Table 6 lists methods and properties of this interface.

Method	**Description**
ID	Gets the ID string for the link.
Description	Gets a string with description of the link.
SourceComponent	Gets the source LC.
SourceElementSet	Gets the source element set (IElementSet).
SourceQuantity	Gets the source quantity (IQuantity).
DataOperationsCount	Gets the number of data operations selected from source input exchange item.
GetDataOperation	Gets the n-th selected data operation.
TargetComponent	Gets the target LC.
TargetElementSet	Gets the target element set.
TargetQuantity	Gets the target quantity.

Table 6 Methods of *ILink* interface

2.4. Computation

Before the computation can start, *Prepare* method must be called on all LCs in the composition. The computation is then triggered by invoking *GetValues* method on a one of LCs. LC may then, in order to compute its result, invoke *GetValues* method on other inter-linked LC, and so on. In OpenMI this is known as pull-driven mechanism.

Important thing is that all *GetValues* calls are done synchronously in a single thread of execution.

After computation finishes, the deployment software must call *Finish* method on all LCs in the composition, so they may for example save the result files.

2.5. Events

The events system is kind of messaging between the OpenMI linkable components and the external tools, and is a substantial part of the OpenMI standard. Events allow the implementation of tools that perform tasks such as a logging, tracing, or online visualization. Linkable components can generate events to which other linkable components or tools can subscribe (using LC's *Subscribe* method). The event listener must implement *IListener* interface, Table 7 shows its methods:

Method	**Description**
GetAcceptedEventTypeCount	Gets the number of event types, which the listener wants to listen.
GetAcceptedEventType	Gets the n-th listened event type (*EventType)*.
OnEvent	Called by LC to send an event to the listener .

Table 7 Methods of *IListener* interface

Events are handled, like the computation itself, synchronously. When event is sent to a listener using *OnEvent* method, the listener grabs the thread's call stack. This allows listeners to implement for example pause functionality, or even listeners are able to cancel the computation by throwing an exception.

3. Distribution of Computation

In this section we introduce basic ideas behind the distribution of OpenMI computation, and discuss the possible use cases.

The OpenMI models can internally utilize any resources on the network, or even perform the calculation on a remote computer; however, there is no standardized way how to combine OpenMI models running on different computers into a single OpenMI composition. The aim of this thesis is development of a framework that will seamlessly allow the distribution of arbitrary OpenMI compliant models to different computers and run them in a single OpenMI composition, using standard OpenMI tools. In the following text we will refer to this framework as *Distributed OpenMI*.

3.1. Concept

Basic idea is to have a framework with the ability to make OpenMI compositions on the local computer including both local models and models running on remote computers using existing OpenMI software and tools. All OpenMI compliant models may be used as remote models and it should be transparent for other models whether they are linked to a remote or local model.

Obviously, there are two types of clients in Distributed OpenMI: the clients who provide the access to their local OpenMI models to other clients; and the clients who access these models.

3.2. Use Cases

There are several situations where it is useful to link the models running on different computers:

- **Provide remote access to a large time series databases**
  The time series databases (storing e.g. sensor data, whether forecast …) may be wrapped into OpenMI linkable components and provided to computational models using Distributed OpenMI. The computation models will only request the data, which they actually need for their computation.
- **Deploy the simulation on the desktop, run it on a dedicated machine**

Client PC may only be a place from which the simulation is invoked and monitored. The computation may run on dedicated high-performance computer where the model actually resides.

- **Link models from different providers without a need for moving data**
  Classical OpenMI approach necessitates moving all input data to a single computer before running the simulation. If the amount of the data is big, the preparation phase may bring significant performance overheads. Distributed OpenMI allows running the models on the computers where the data is available. Only the data necessary for the linkage will be transferred between the computers.
- **More effective usage of simulation software licenses**
  The commercial simulation software is very expensive, so the number of installations customers can use is limited. It is inefficient to have all simulation software installed on a single computer.

All previous situations may be solved in other ways; however, our concept enables developers to use a simple framework to achieve these goals and protects them from implementation of a complex distributed functionality.

3.3. Remote Procedure Call

OpenMI is an object oriented system, the computation consists of method calls between the objects representing inter-linked models and thus a natural approach how to distribute the computation is to use *Remote Procedure Call* (RPC). There are many implementations of remote procedure calls; the paper [Bir83] gives general background about implementation of RPC. RPC will be used to invoke the methods of original *Linkable Component* (LC) on a particular client.

Our distributed system aims to be OpenMI compliant, so any changes in the existing OpenMI models providing the remote access to other models are not acceptable. The only entity, which can be linked to existing LC, is other LC. This naturally implies that we have to encapsulate the RPC client functionality into a special LC, which can be seamlessly linked to existing LCs. In our system, we introduced *Remote Linkable Component* (RLC) to proxy the method calls to original LC on a remote side.

On the other hand, it is useful to introduce additional layer between original LC and RPC server subsystem. In Distributed OpenMI the *Model Provider* (MP) component is used to receive remote calls from the RPC subsystem and to forward them to original LC. Moreover, this layer allows us to implement functionality like authorization (see Section 4.3), direct local model linkage and piggybacking (see Section 7).

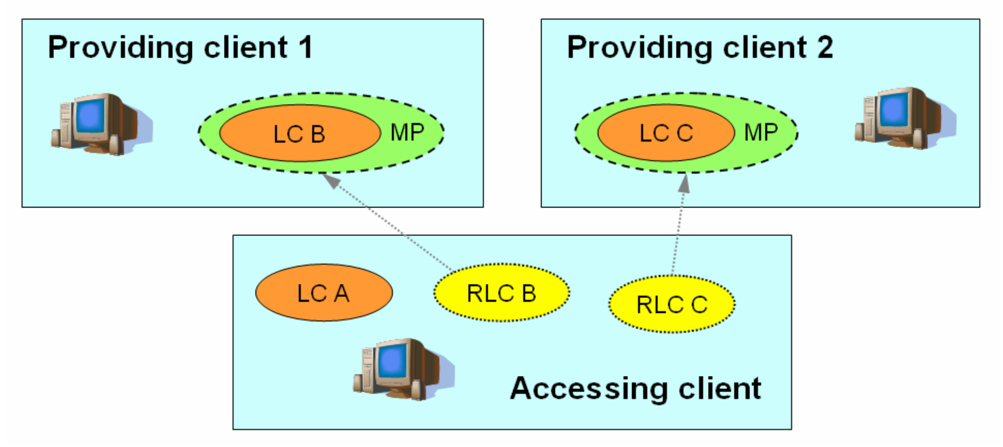


Figure 1 Basic concept of the remote access to the models

3.4. Model Provision and Access

OpenMI models are described using OMI files, thus the Distributed OpenMI clients, who provide the access to their local models, must able to register these models using the OMI files. On the other side, the accessing client must have special OMI file holding all the information needed by Distributed OpenMI system to connect to the providing client and access a particular model.

On the start, the deployment software on accessing client instantiates RLC using the information from the OMI file. This in fact means that the Distributed OpenMI system is started in the deployment software process. RLC connects to the remote providing client and requests it to instantiate LC for the original model (using a registered OMI file). After that, the providing client creates MP around LC, and registers it to RPC subsystem in order to receive the remote method calls. The instantiation of a provided model must be done on demand because single client can provide large number of models and their pre-instantiation could bring significant memory overheads. After MP is created, the accessing client receives a RPC-reference to it, which is attached to the RLC. Now RLC is ready to proxy calls to remote LC.

In Section 4 there is more detailed information about all mentioned components and processes.

3.5. Platform

Current OpenMI release (version 1.2.0) is targeted to Microsoft .NET Framework 2.0 (an implementation of CLI, as defined in [Ecm06]). This naturally implies that Microsoft .NET Framework 2.0 is the platform, on which the Distributed OpenMI system should be running. Although the OpenMI Standard has earlier been released for the Java platform as well, the support for Java has been stopped recently by OpenMI Association, which is the responsible authority maintaining the OpenMI Standard. This is reason why a formerly proposed Java implementation of the client has been abandoned.

4. Making Linkable Components Remotable

In this section, we describe components and processes taking part in the preparation and execution of a distributed OpenMI composition.

4.1. Remote Linkable Component

The *Remote Linkable Component* (RLC) is a substantial part of the Distributed OpenMI. It implements the OpenMI's *ILinkableComponent, IPublisher, IManageState* and *IDiscreteTimes* interfaces, and is used in the OpenMI composition to proxy method calls of these interfaces to original *Linkable Component* (LC) instantiated in a different process, eventually on a remote computer. Additionally, RLC is the initiator of the access to a remote model (see 5.4), and incorporates the authorization (see 4.3) and caching (see 7.2) functionality.

If the RLC is instantiated by the OpenMI deployment software as a result of opening the OMI file (as described in 2.2), we talk about *explicit instantiation*. If RLC is instantiated as a result of model linkage, we talk about *implicit instantiation* (see 4.6).

4.2. Model Provider

Model Provider (MP) is a wrapper around original linkable component, which enables invocation of its methods from a remote side over RPC. For each *ILinkableComponent* interface method MP has an equivalent method. In our implementation, the *IModelProviderRemote* interface represents all MP's methods which may be called from a remote side (*remotable methods*).

There are more reasons why it is useful to introduce this additional layer. Generally, for security reasons, it is not a good idea to allow the direct access to underlying LC from RPC server subsystem. Moreover, some RPC subsystems have restrictions about the objects receiving remote calls (e.g. in Microsoft .NET Remoting the server objects must be inherited from *MarshalByRefObject* object, as described in [Net07]). Such restrictions may not be fulfilled by a particular linkable component implementation, which could even by legacy and cannot be changed. Additionally, MP is useful to implement piggybacking performance optimizations (see 7.2).

Note that each remote model in Distributed OpenMI is uniquely identified by GUID (Globally Unique Identifier, as defined in [Rfc4122]), which is stored in both RLC and MP. This GUID is for example useful for direct model linkage (see 7.1).

4.3. Authorization

Using MP, we could also supply simple an RPC-independent authorization mechanism which prevents unauthorized remote access to provided LC. In our implementation, each MP stores a 16 byte long unique authorization ticket (*Auth*). This ticket is generated by a cryptographically safe random number generator (in our implementation using *RNGCryptoServiceProvider* provided by .NET, see [Net07]). When calling any remotable method on MP, the caller must supply exactly the same authorization ticket in a special input parameter. If supplied ticket does not match, an exception is thrown. During the remote model access handshake, the MP's authorization ticket is provided to corresponding RLC, which stores it internally and uses it for each call to remote MP.

4.4. Object Serialization

In OpenMI there are several objects, described via interfaces, which are passed to or returned from methods of a linkable component. These objects are state-less and residually independent according to the OpenMI standard, thus it is useful to transfer them by-value over the process or computer boundary.

The implementations of the objects may vary, and it is not ensured that a particular implementation may be serialized by the RPC subsystem (e.g. in .NET Remoting all serializable objects must be flagged with *[Serializable]* meta-data attribute, as described in [Net07]). This is the reason why these objects are converted to a special serializable representation before they are passed to the RPC subsystem. The serializable representation stores all the information from the original object accessible via the standard OpenMI interfaces. The values of internal attributes are lost in this process, what is not a problem since linkable components should only provide the data using the well-known interfaces according to the OpenMI standard.

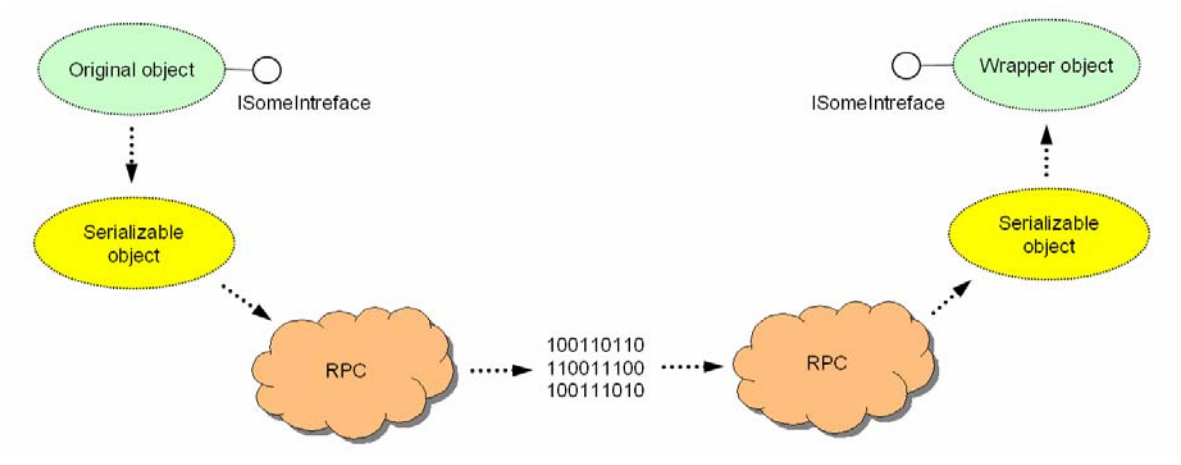


Figure 2 Serialization and deserialization of OpenMI objects

After the serializable representation of an object reaches its destination on the remote side, a wrapper object implementing particular OpenMI interface is created from it. This object is then passed to original LC (RLC – MP – LC way), or returned as result of RLC method call (LC – MP – RLC way). The wrapper object emulates the state of the original object in the way that all its properties and methods return the same values (for same input parameters, eventually). Here we assume with respect to the OpenMI standard that original object does not change its state in the time. The only exception is *IDataOperation* object, as explained in Table 8. Note that we cannot instantiate the original object on a remote side because its implementing type may not be present there. The transfer of an assembly implementing specific type to a remote host is not possible due to security reasons.

The conversion to the serializable representation and creation of the wrapper is handled transparently by RLC and MP components. Table 8 lists objects for which this is needed and comments the non-trivial cases:

Object	**Comments**
IArgument	
IDataOperation	Problematic are *Initialize* and *IsValid* methods. *Initialize* method may change the state of the instance. The resulting value of *IsValid* method for all combinations of input parameters could be calculated on the original object, but for that (*number of input exchange items*) x (*number of output exchange items*) x (*number of data operations*) calls to that object would be needed, what is not acceptable. In our implementation, the wrapper object has a reference to owning RLC, and calls to both *Initialize* and *IsValid* methods are proxied to original LC on the remote side. MP has *DataOperationInitialize* and *DataOperationIsValid*

	methods which call the corresponding methods on the original *IDataOperation*.
IDimension	Serialized as part of *IQuantity*, not directly.
IElementSet	
IEvent	References LC which generated the event. The model GUID of that LC is used in the serializable representation. After deserialization wrapper object uses that GUID to find correct LC in its address space (if such exists).
IInputExchangeItem	
ILink	References source and target LCs. Model GUIDs are stored in the serializable representation. After deserialization the wrapper object uses that GUID to find correct LCs in its space (it is guaranteed they exist, see 4.6).
IOutputExchangeItem	
IQuantity	
ISpatialReference	
ITime	Itself has no properties and is just base interface for *ITimeSpan* and *ITimeStamp*. The serializable representation stores the information about which interface is actually implemented and after deserialization the wrapper object implements same one(s).
ITimeSpan	Serialized as *ITime*
ITimeStamp	Serialized as *ITime*
IUnit	Serialized as part of *IQuantity*, not directly.
IValueSet	Base interface for *IScalarSet* and *IVectorSet* interfaces. The serializable representation stores the information about which interface is actually implemented and after deserialization the wrapper object implements same one(s).
IVector	Serialized as part of *IVectorSet*, not directly.

Table 8 OpenMI objects transported over network

The OpenMI Standard also defines several enumerations (*DimensionBase*, *ElementType*, *EventType* and *ValueType*). All of them are based on a 32-bit integer, thus the serialization is trivial.

4.5. Object Deposit

Although the OpenMI Standard has been developed quite recently, it is not designed to make the remote access to linkable components simple. One of the issues is that several LC's (and *IDataOperation*) methods expect as input parameters objects provided earlier by other methods. If just an identifier of these objects could be used, the situation would be much easier. Passing the wrapper object instead would defy the OpenMI Standard and could cause errors since a linkable component may depend on passing of correct objects. Table 9 lists problematic methods and the objects they need (*fixed objects*):

Method	**Passed Object(s)**
AddLink	*IElementSet, IQuantity, IDataOperation*
IDataOperation.IsValid	*IInputExchangeItem, IOutputExchnageItem, IDataOperation*
HasDiscreteTimes	*IElementSet, IQuantity*
GetDiscreteTimesCount	*IElementSet, IQuantity*
GetDiscreteTime	*IElementSet, IQuantity*

Table 9 Fixed objects in OpenMI

In our implementation, the *Object Deposit* component has been developed to store fixed objects produced by a linkable component. With each fixed object (including its serializable representation and wrapper) there is a GUID associated. Whenever LC produces an instance of a fixed object, MP looks into the object deposit whether that object is already present there. If yes, the serializable representation of that object reuses a same GUID. If not, a new GUID is generated for the object and saved to the object deposit (together with object itself).

When a problematic method is called on RLC, it is expected that input parameters are our wrapper objects, according to the OpenMI Standard. RLC takes GUID from the wrapper object, and passes only that to remote MP. The MP finds the corresponding fixed object in its object deposit, and passes it to original LC (as depicted on Figure 3).

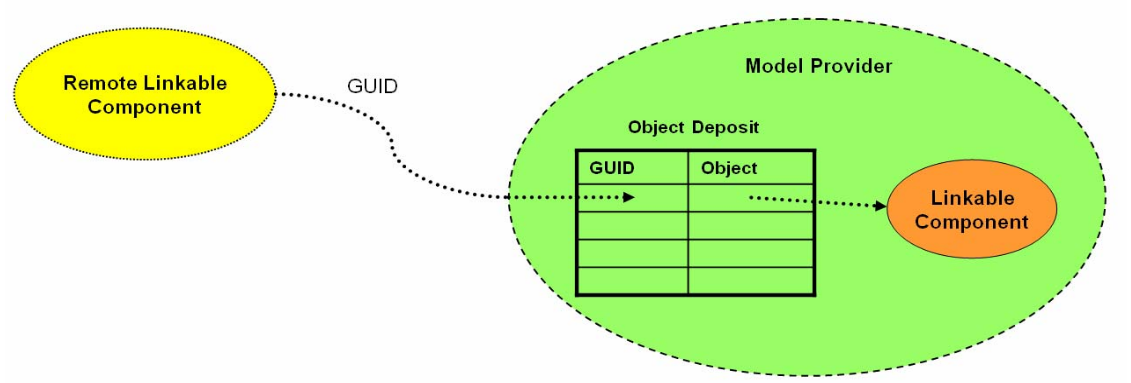


Figure 3 Usage of object deposit

Internally, the object deposit consists of two complementary hash-tables, one has GUID as the key and reference to the fixed object as the value, second has reference to the fixed object as the key and GUID as the value. All object deposit operations have O(1) amortized complexity.

4.6. Model Linkage

In OpenMI, to link two models, the deployment software calls *ILinkableComponent*'s *AddLink* method on both source and target LC. As parameters of this method, the references to both LCs are supplied. This means that after *AddLink* is called, one LC can invoke methods of other LC, and vice versa.

If one of the inter-linked models is RLC, simply the reference to RLC is supplied. However, the *AddLink* method still must be called on the original LC on the remote side, supplying a reference to some there local LC which would represent the inter-linked LC. The natural approach solving this situation is the instantiation of RLC on remote side, which would proxy the calls to locally inter-linked LC. In Distributed OpenMI this is known as implicit RLC instantiation. Of course, MP has to be created around locally inter-linked LC, enabling it to receive calls from remote side, as depicted on Figure 4.

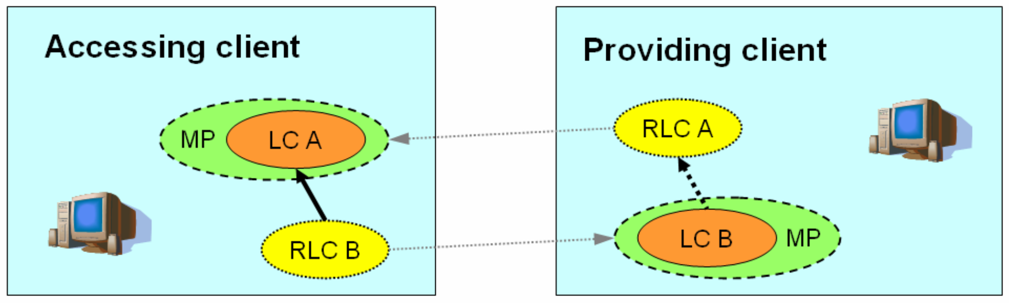


Figure 4 Link between local and remote model

The creation of MP and implicit RLC instantiation on the remote side takes place during the local *AddLink* call. Note that the remote model on a providing client side now has the opportunity to perform callbacks to accessing client, what is one of the requirements posed on a RPC communication protocol.

There are some special cases in the model linkage, which may be handled different way to achieve better performance (see 7.1.2).

4.7. Event Callback

Event handling is an important part of the OpenMI standard and our system aims to support it entirely. Each LC is producer of events, and any component implementing

OpenMI's *IListener* interface may subscribe to LC in order to receive its events. This approach must be preserved for RLCs as well.

When an event listener subscribes to RLC using *Subscribe* method, the subscription must also be done on the remote side. After that, if LC on the remote side produces an event, the event must be send to RLC using a callback. The OpenMI Standard defines that events are handled synchronously (i.e. LC may call the *IListener*'s *OnEvent* method only during the execution of some method and the computation is blocked until *OnEvent* returns, as depicted on Figure 5). This implies that our system must handle events synchronously, as well. If we used piggybacking mechanism to pack occurred events together with result of some later call to remote LC, the state of the OpenMI composition would never be the same as the state when the event really occurred. This could cause strange behavior.

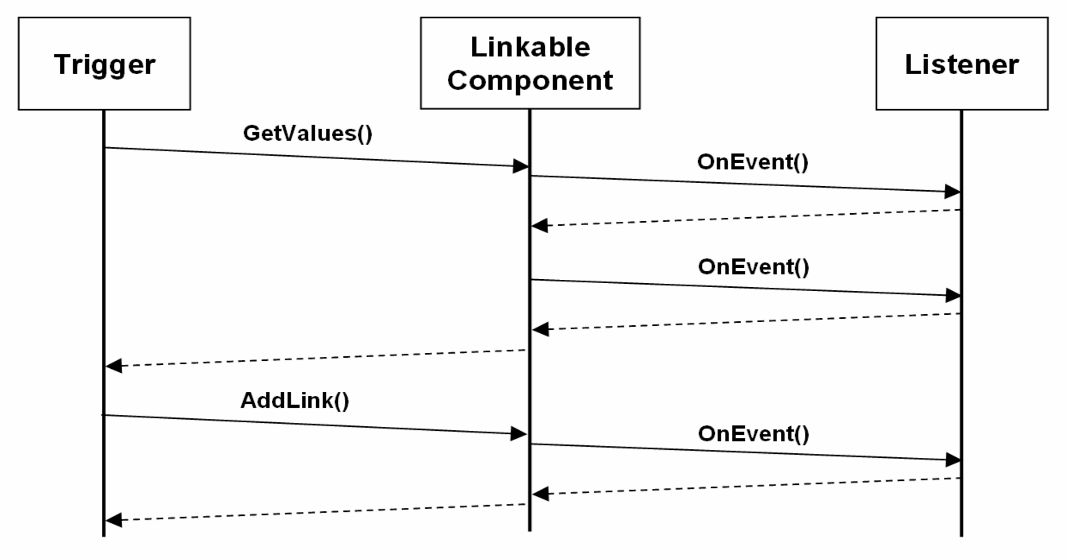


Figure 5 Synchronous event handling

In our implementation, the *IEventReceiver* interface is used as a callback interface for event dispatching. RLC implements this interface and registers itself to the RPC subsystem to be able to receive event callbacks. The RPC-reference to RLC is sent to MP during the event subscription request.

4.8. Threading

The OpenMI Standard defines that composition preparation and computation is run within a single thread of execution. There are models and tools which depend on this aspect of OpenMI, for example:

- **Graphical utilities**

  For example, on Microsoft Windows operating system, after a window is created, all manipulation with it (using operating system's handle) must be done within the same thread of execution that created the window (described in [Win07]).

- **Legacy models**

  For example, industry standard river hydraulics model *MIKE 11* (developed by *DHI Water, Environment and Health*) utilizes several COM (*Component Object Model*) components with STA (*Single Thread Apartment*, see [Win07]) model. This necessitates that all method calls to LC must be done within a single thread of execution, because the methods internally utilizes mentioned COM components (explained in [Win07]).

Because Distributed OpenMI aims to support all OpenMI compliant models, we must ensure that all LC methods are called within a single thread of execution. In fact, our task is to spread single-thread call stack to multiple processes, possibly on different computers. To achieve that, for each provided model, there must be a reserved thread on which all calls to LC will be executed. Moreover, all LCs from a single composition on a single client must use the same thread to enable the direct model linkage, which makes possible that LCs call directly each other (see 7.1.2). The OpenMI Standard does not forbid cycles in calls between LCs in the composition. In our distributed system this means that nested RPC calls must be dispatched on a causal thread waiting for a result of another pending RPC call.

Possible solution is to assign single thread on the providing client for all models belonging to a same composition. MP would then marshal the LC calls to that thread. The RLC would need to send the RPC calls asynchronously, and then grab the calling thread so that MP could marshal the nested calls again to that thread.

Although the solution of threading issues could be implemented on the level of RLC and MP as outlined, the idea to separate this quite general problem became more interesting. From the discussion in Section 6 the need for implementation of a proprietary RPC protocol arises. Our threading issues (namely reserved worker thread for dispatching of remote method calls and execution of nested remote calls on the blocked causal thread) may be solved transparently on the RPC level, bringing a clean and reusable solution. The decision to go this way has been made. In the following paragraphs we describe how threading issues have been solved using the features of new YA-RPC protocol. In Section 6.3 we explain how these features were actually implemented.

4.8.1. Reserved Worker Thread

YA-RPC protocol has the possibility to assign *Single Thread Worker Queue* (STW) for specific server objects registered to the RPC subsystem. When a remote method call is received, the request is queued to STW and executed after STW thread has nothing to do. The first idea is to associate one STW with each MP on a providing client and perform the calls to underplaying LC directly on the STW thread. However, as mentioned, the MPs for LCs belonging to a same OpenMI composition must share the same thread, thus must share same STW. Unfortunately, the fact that two LCs fall into a same OpenMI composition is first determined when they are linked together – in time when they are already initialized and STW is already assigned. Two LCs from one providing client, accessed even by a same client, may of course be completely independent, thus assignment of one STW to all MPs is not possible.

In our implementation we assumed that a single OpenMI composition is identified not by the model linkage, but by the thread under which the models are instantiated on the accessing client. This perfectly fits our concept of a single thread call-stack spreading, because if the OpenMI deployment software instantiates two RLCs in two different threads, it cannot suppose that corresponding LCs on remote side shares a same thread. When RLC is instantiated explicitly, it sends globally unique thread identifier (16-byte GUID) together with model access request to the remote providing client. The providing client looks whether STW for this unique identifier already exists; if yes it reuses the STW, if not it creates a new one.

As the thread identifier we could not simply use operating system's identifier or handle, since it is not guaranteed to be globally unique. Our globally unique identifier is stored in the thread local storage (abstracted using .NET *[ThreadStatic]* meta-data attribute, as described in [Net07]).

Because there is some performance overhead associated with usage of STW and not all models necessitates the execution under a single thread, the providing client have the opportunity to turn this feature off.

4.8.2. Nested Remote Calls

Other nice feature of YA-RPC is that it has the option to track the nesting of remote calls and execution of nested remote calls on a blocked causal thread. It means that if the node A sends a synchronous (i.e. blocking) remote call to node B (*causal call*), and node B sends another call (*nested call*) back to node A during the execution of the causal call, the nested call is dispatched on the A's blocked thread (*causal thread*). This feature solves completely our problems, since MP executes methods of LC directly without additional marshaling layer.

Note that STWs for model providers are necessary only on providing clients, accessing clients do not need them because the initiator of the computation is some (causal) thread on the accessing client, and callbacks to its MPs are executed on that blocked causal thread. Although, if LC on the providing client asynchronously (out of the computation call-stack) invokes some method on accessing client's LC, that call is executed on a thread-pool thread, and that is perfectly correct.

5. Integrating Server Model

Distributed OpenMI aims to distribute the computation in the internet wide environment (WAN), and not only in the intranet (LAN). However, many computers and networks in today's internet are hidden behind firewalls and NAT (*Network Address Translation*, described in [Rfc3022]). The configuration of firewalls and routers allowing connection to protected networks may be apparently a problem, e.g. for security reasons, or just for a user inability. In the server-less model, each providing client would have to make configuration changes in order to use Distributed OpenMI. This could limit the number of model providers and make the commercial expansion of the system impossible.

This led to an idea of one central server, which would be the only place where the firewall or router configuration settings need to be changed. Silently we pose another requirement to selected communication protocol – the ability to perform callbacks without the need of server-initiated connection. If the communication protocol does not have this capability, the central server is meaningless.

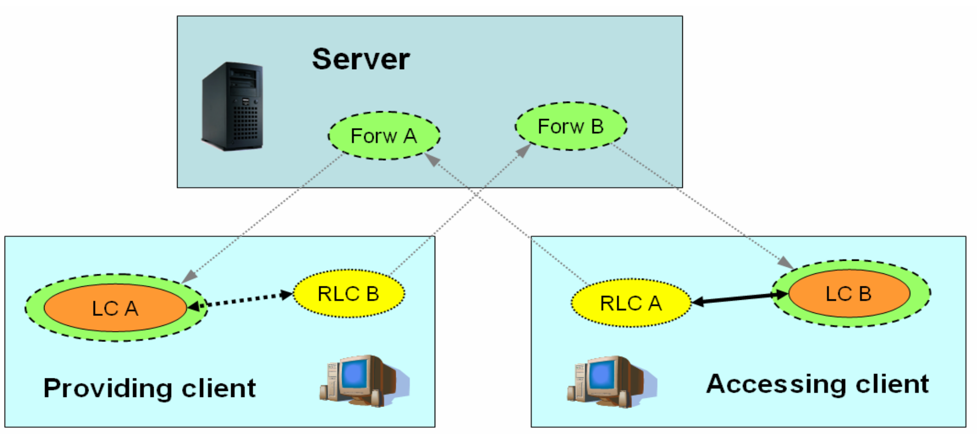


Figure 6 Integrating server model

5.1. Clients Management

The presence of a central server in Distributed OpenMI brings also the possibility to add features which would not be possible otherwise. Server may maintain the list of clients, serve their authentication, and maintain the list of provided models… For the communication between the clients *Remote Procedure Call* (RPC) has been proposed. Natural approach how to publish these server features is also to use RPC. In our implementation, access to the server is abstracted using the *IServer* interface. Table 10 summarizes its methods.

Method	**Description**
GetGuid	Gets the unique identifier of the server.
GetServerInfo	Gets the basic information about the server.
RegisterUser	Registers the client to the server.
UnregisterUser	Remove registration of the client from the server.
LogOn	Performs logon to the server.
RenewAuth	Renews logon on the server.
LogOff	Logs off from the server.
ProvisionStart	Starts the model-provision mode.
ProvisionStop	Stops the model-provision mode.
ProvisionAddModel	Add a provided model.
ProvisionUpdateModel	Updates a provided model.
ProvisionRemoveModel	Removes a provided model.
GetAllUserNames	Gets the client-names of all providing users.
GetAllModelNames	Gets the names of all provided models for a specific user.
GetModelInfo	Gets the information about a specific provided model.
AccessModel	Initiates the access to a remote model.

Table 10 Methods of *IServer* interface

To be able to use most of the server methods, the client must be logged on using the client-name and password (*LogOn* method). If the client does not have an account, it can create a new one using *RegisterUser* method. In our prototype implementation the registration always succeeds, what in fact means that the anonymous user accounts are allowed. One can imagine that, in the commercial environment, the registration may be contingent on some approval (e.g. based on a received payment). Client-names must be unique within the single server, of course. In our implementation the server persists the list of registered users regularly to a data file.

After the registration and logon to the server, the client is able to provide access to local models, to browse the other providing clients, browse their provided models and also access the models. In the source code of our implementation we sometimes refer to "client" as "user" (because clients are kind of users of the server); however in the following text we will only use the term "client".

Note that every Distributed OpenMI server is identified by a globally unique identifier (16-byte GUID). This identifier is used by clients to determine that specific server is already connected. The server hostname cannot be used for this purpose, because more hostnames may point to single host.

5.2. Remote Calls Forwarding

The most important role of the server is that it forwards remote method calls from a one client to another. This regards the calls to *Model Provider* (MP) and event callbacks to *Remote Linkable Component* (RLC). The forwarding works the way, that the server registers objects with same interface (*forwarder*) to the RPC subsystem. When a client sends a remote call to the forwarder, it internally sends another RPC call to the original object on the target client. When the latter call finishes, the forwarder returns the result to the calling client.

The forwarding of remote calls must be asynchronous – forwarder's method must just schedule the remote call to a target object, and return immediately. After the method on the target object returns, the call to forwarder may also be finished. If the call to forwarder's method would block the thread on the server until target client object's method returns, a malicious client could simply "hijack" a high number of server's threads, what in fact is kind of *Denial of Service* (DoS) attack. The ability to asynchronously forward remote calls poses a new requirement on the selected communication protocol (see Section 6).

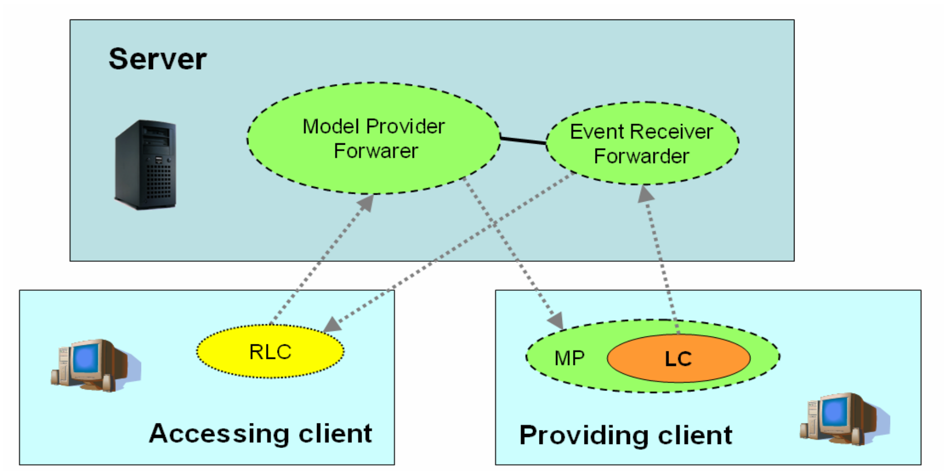


Figure 7 Model provider and event receiver forwarders

On the Distributed OpenMI server there are two types of forwarders: *Model Provider Forwarder* (based on *IModelProviderRemote* interface) and *Event Receiver Forwarder* (based on *IEventReceiver* interface). First one forwards the remote calls in the RLC-Server-MP direction, second one in the MP-Server-RLC direction. The concept of remote call forwarding does not affect introduced concept of clients much. The clients simply send all calls to the server, instead to the target clients.

Additionally, in our implementation, during the forwarding the server checks the authorization ticket for the model provider (described in Section 4.3) and tracks the event subscription process in order to forward the events only where needed (see Section 7.3).

5.3. Model Provision

Before the client can enter the provision mode, it must logon to the server. Using *IServer* interface's *ProvisionStart* method the client enters the provision mode. As parameter of this call the client must supply RPC-reference to *IProvisionCallback* callback, which is used to initiate access to the model on demand.

The providing client process then waits for incoming requests. The client periodically calls server's *RenewAuth* method to refresh the logon. This is necessary since the logon timeouts after a period of time to prevent dead clients to be considered alive.

5.4. Model Access

After explicit instantiation of RLC in the OpenMI deployment software's process, Distributed OpenMI connects and logons to the server using credentials stored in corresponding OMI file. After that, RLC calls server's *AccessModel* method. Server finds requested providing user, and using callback requests it to provide the access to the model (i.e. calls *IProvisionCallback* interface's *ProvideModel* method). RPC-reference to event receiver forwarder is sent as part of this request.

As described in Section 3.4, the providing client instantiates and initializes the linkable component and creates a MP for it. Reference to MP is sent back to server as a result. The server then creates a model provider forwarder and sends its RPC-reference back to the accessing client as a result. Now the client is able to call the methods of original LC on a remote side, and the event system is ready to work.

6. Communication Protocol

Selection of a right RPC communication protocol was one of the biggest challenges during the development of Distributed OpenMI. The architecture of the system poses very specific requirements on the selected protocol. In this section we compare available protocols with respect to our requirements, and explain why we decided to implement a proprietary protocol.

6.1. Requirements

The following list summarizes all the requirements posed on the communication protocol by our Distributed OpenMI concept. There are other requirements not explicitly pronounced, which we silently consider as a matter of course (e.g. reliable delivery, stable release, synchronous method calls, serialization of complex data types …etc).

- **Callbacks on client initiated connection**
  As mentioned many times, the server must be able to invoke methods on the clients. The client-server connection must be client-initiated to deal with firewalls and NAT (explained in Section 5).
- **Asynchronous remote calls and server methods dispatch**
  The RPC must have opportunity to invoke a remote method call without the need of blocking the calling thread. When the remote method finishes, a supplied callback has to be called by the RPC subsystem. Similarly, a server object must have the opportunity to return immediately when its method is called and be able to notify the RPC subsystem about the completion and return value. These two features are necessary for the asynchronous remote call forwarding on a server (as described in Section 5.2).
- **Free availability for Microsoft .NET Framework**
  Distributed OpenMI may become either commercial or open source software. In both cases the dependence on a commercial RPC protocol would be very unpleasant. An open source or SDK library is the desired choice, for Microsoft .NET Framework of course.
- **High performance**
  The distribution of OpenMI computation must have as small performance overhead as possible, and RPC must be fast enough to achieve this

objective. Note that Distributed OpenMI may also be used in intranet environment, where network latency is insignificant – here the performance of the RPC subsystem may be a bottleneck.

- **Internet wide usability**
  The clients connected to the server may be in different networks of internet. This limits the choice of RPC's underlying protocol to widely extended and supported TCP/IP protocol.

6.2. Protocol Comparison

In Table 11, we list the most widely used RPC protocols, and show whether they implement the necessary features.

	Callbacks	Asynchronous call forwarding	Availability	High performance	Internet wide usability
CORBA [Omg04]	✓	✓		✓	✓
DCOM [Dcom98]	✓		✓	✓	
Java RMI [Java03]				✓	✓
.NET Remoting [Net07]			✓	✓	✓
Web services [Web04]			✓		✓
XML-RPC [Xml99]			✓		✓
FastRPC [Fast]				✓	✓
ONC RPC [Rfc1831]			✓	✓	
DCE/RPC [Dce97]	✓			✓	✓

Table 11 The comparison of RPC protocols

The summary is that for our purpose no existing RPC protocol is available. This led to the decision to implement a proprietary protocol, which would incorporate all the necessary features. Additionally, the proprietary protocol enables us to transparently solve threading issues on the RPC level (as described in Section 4.8) and is ready for a future extension (e.g. Java implementation, encryption, transparent failover …etc). The new protocol was named *YA-RPC*.

6.3. YA-RPC

YA-RPC stands for *Yet Another Remote Procedure Call*. It is a simple binary communication protocol which has been developed from the scratch. As underlying transport protocol, the *Transmission Control Protocol* (TCP) is used (described in

[Rfc793]). The choice of TCP was very natural – it is simply the most widely supported internet protocol with reliable stream delivery. The remote method calls and return values are packed into messages, which are transported using TCP over the network. The first implementation of YA-RPC has been done for Microsoft .NET Framework.

In this section, we only show the most important features of YA-RPC protocol. The technical details are beyond the scope of this document and may be found in the YA-RPC documentation.

6.3.1. Server Objects and Methods

YA-RPC uses object-oriented approach, i.e. remotable methods are defined on objects (*server objects*). Each server object must implement *IYaRpcRemotable* interface. Before an object may receive the remote method calls, it must be registered to the YA-RPC subsystem. The server objects are identified by a globally unique identifier (16-byte GUID), which is supplied during the registration. The methods of server objects are identified using 32-bit signed integer. When a client performs the remote call, it must provide both the object GUID and method ID. The server object methods may have unlimited number of input parameters, and unlimited number of return values.

During the dispatch of the remote call, the YA-RPC subsystem first calls *IYaRpcRemotable*'s interface *GetMethodDefinition* method on the server object. Based on the result of this method, either the *ExecuteMethod* or *BeginExecuteMethod* method is later called on the server object to perform the job. Additionally, *GetMethodDefinition* method is used by the YA-RPC subsystem to determine the types of input parameters for a method. These types are used to deserialize the message, so that the parameters may be supplied to the method in fair form.

To receive the remote calls, the server must either be listening on a specific network interface and TCP port, or it must be connected to another remote host.

6.3.2. Remote Calls

To perform the remote call over YA-RPC, the caller must have open connection to remote host, server object GUID, method ID and the array of method parameters, whose types must exactly match the types defined by *GetMethodDefinition* method on the server, so that these parameters are deserialized correctly. Moreover, the caller supplies array of types defining the return values of the method. These types are used to deserialize the return message received from the server after remote call finished. Of course, the return values of the remote method must exactly match these types.

The types of parameters and return values supplied by the server object and the caller define the RPC communication interface. For this purpose, other RPC protocols use more user-friendly approaches like *Interface Definition Language* (IDL, described in [Omg04]), which however may be quite complicated to implement. Fortunately, YA-RPC provides helper methods which automatically generate the method definitions for a supplied type (implemented via .NET Reflection).

The remote call can be either synchronous (i.e. blocking) or asynchronous. In the second case, the caller supplies a callback delegate which is called after the remote call finishes (or fails). The caller can also specify a timeout for the remote call, after which the unfinished call fails.

As described earlier, remote calls may be sent both from the client to the server, and from the server to the client. In this context, the client is understood as the initiator of the connection.

6.3.3. Serialization

YA-RPC serializes objects to a binary representation with little-endian byte order. Common data types (*Boolean*, *Byte*, *SByte*, *Int16*, *UInt16*, *Int32*, *UInt32*, *Int64*, *UInt64*, *Single*, *Double*, *Decimal*, *String*, *DBNull, DateTime*, *Guid*) and arrays of such types (and arrays of arrays, recursively) are serialized automatically. The serialization of complex data types is also possible, however these objects must implement *IYaRpcSerializable* interface to control the serialization process by themselves. Generally, the serialization process is platform independent, thus future migration to Java and potentially other platforms is possible.

6.3.4. Asynchronous Remote Calls Forwarding

YA-RPC supports both asynchronous remote calls, and asynchronous server methods dispatch. These features are necessary for an implementation of asynchronous remote call forwarding.

As discussed earlier in this chapter, the remote call can be send asynchronously. Similarly, the server object may using *IYaRpcRemotable* interface's *GetMethodDefinition* method specify, that specific method is asynchronous. In such case the YA-RPC subsystem calls *BeginExecuteMethod* to invoke the method. After method's job is finished, a supplied callback must be called by the server object in order to finish the remote call in YA-RPC.

6.3.5. Performance

During the development of YA-RPC, the performance was one of the key aspects taken in the mind. Following list summarizes used methodologies which helped with achievement of this goal:

- **Low-level TCP protocol**
- **Binary serialization**
- **Asynchronous socket operations**

  For the network subsystem manipulation, our implementation uses socket interface provided by .NET Framework libraries, which internally utilizes *Windows Sockets* (described in [Net07], [Win07]). With Windows Sockets the best performance, throughput and stability is achieved using the asynchronous model, because it internally utilizes *Windows NT I/O Completion Ports* (IOCP), as described in [Jon02]. YA-RPC adopts this model.
- **Intelligent growth and shrink of receive buffer**
- **Remote call forwarding with no additional context switch**

  When dispatching an asynchronous remote call, the invocation of the asynchronous method (i.e. the send of forwarded remote call) is done directly in the IOCP thread, thus no additional context switch is needed. This is a desired behavior for high performance server applications.
- **Multiple listening sockets**

  The server listens on several sockets at the time, with pre-prepared accepting sockets. This is useful in order to serve high number of newly opened simultaneous connections. The number of listening sockets is configurable.

6.3.6. Execution of Nested Calls in Causal Thread

If a client invokes synchronously remote call on a server (*causal call*), and the server sends a callback to the client as part of the execution of that call (*nested call*), the callback is dispatched on the client's blocked thread (*causal thread*). Naturally, during execution of one nested call other remote call may be send … and so on. It is desired that both asynchronous remote call and asynchronous call dispatch preserve this behavior (i.e. remote call forwarding will not break this approach).

In YA-RPC each remote call is identified using 16-byte GUID. The message representing the remote call contains the GUID of the causal call send from that host, if such exists. When dispatching the remote call, the YA-RPC subsystem looks whether there is some pending synchronous call with that GUID, and if so, uses the blocked thread to execute the call.

Crucial is how to obtain the GUID of the causal call. Each YA-RPC host must track the information, which remote call caused the invocation of other remote call. In the situation where communication between only two YA-RPC hosts is done is simple – in fact we only need to acquaint the thread dispatching the remote call with the GUID of that call. This GUID must be saved to some location accessible only by the dispatching thread – in YA-RPC we used Windows *Thread Local Storage* via .NET Framework *[ThreadStatic]* attribute for this purpose (described in [Net07] and [Win07]).

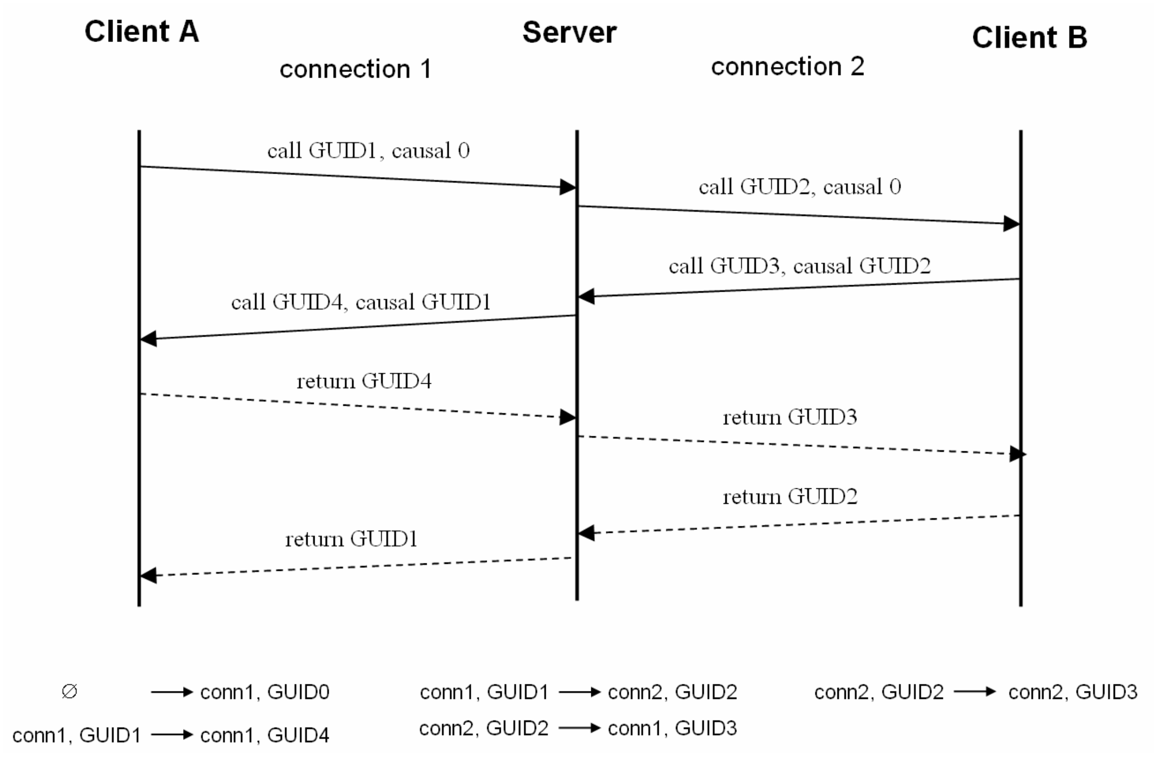


Figure 8 Execution of nested method calls in YA-RPC

Unfortunately, if the communication is done between more than two hosts, the situation becomes quite more complicated. An example of such case is depicted on the Figure 8. Because the dispatch of the call GUID3 on the server is nested to the call GUID2, it is executed on server's thread which waits for result of the call GUID2. During the dispatch of call GUID3, other remote call GUID4 is sent to client A. This call is nested to the call GUID1, but the thread local storage would say GUID3 is the causal call. To avoid this problem, each host must track more information than only "GUID of currently dispatched call". For this purpose YA-RPC uses special data structure named *Call Stacks*. This structure contains remote-call-stack for each group of nested remote calls. The each record of such call-stack contains remote call GUID and the reference to connection on which the call has been made. The remote call stack tracked by each host is depicted on the bottom part of the sequence diagram. To find the GUID of the causal call, the YA-RPC only needs to iterate thru the call-stack in bottom-down order, and find the topmost call from the connection, where nested call should be sent.

Note that each host tracks only remote calls between direct neighbors and not all remote calls in a whole distributed system. Such kind of tracking would be needed in case we wanted to support execution of nested calls in a causal thread for cyclic remote call graphs. Distributed OpenMI uses only one central server, ensuring acyclic topology of the remote calls, so currently there is no reason why to implement such feature.

6.3.7. Single Thread Worker Queue

The server object has opportunity to specify a worker on which the dispatched remote method call should be executed. This may be done using *IYaRpcRemotable* interface's *GetMethodDefinition* method. The worker is represented using *IYaRpcWorker* interface.

YA-RPC currently ships with only one implementation of this interface – *YaRpcSingleThreadWorker* (single thread worker, STW). Each instance of this worker has one thread reserved for the work. When STW is requested to execute some work item, it internally queues that work item. Queued items are executed on the underlying thread in the FIFO order.

6.3.8. Proxy and Stub Helpers

To simplify the usage of the YA-RPC subsystem, there are two generic abstract classes: *YaRpcProxy<T>* and *YaRpcStub<T>*. The generic type parameter *T* identifies the type which is used as RPC communication interface (i.e. the method definitions are read from it). The remotable methods of that type must be marked with *[YaRpcMethod]* .NET meta-data attribute.

To implement a synchronous proxy class to a remote server object, one can inherit the proxy class from *YaRpcProxy<T>*. The implementer must then provide the implementations for proxy methods, which actually send the remote call. To simplify that process *YaRpcProxy<T>* provides some helper methods. One can note that this is not the most convenient way how to get a proxy object, since .NET Framework provides the opportunity to dynamically generate the IL code (defined in [Ecm06]), which might be used to generate the implementations of proxy methods. Unfortunately, the generation of dynamic type is quite complex to implement and was beyond the scope of our work. However, it is possible to extend YA-RPC with this feature in future.

Analogously, to implement server object, the implementer may inherit server object from *YaRpcStub<T>* class. Thereafter, when a remote call is dispatched by that object, *YaRpcStub<T>* uses internally .NET Reflection to find the corresponding method in derived class, and invokes it automatically.

7. Performance

The network communication is surely the main factor affecting the performance of Distributed OpenMI. Since the amount of data transferred between the *Linkable Components* (LC) and external tools typically is not big, and current network subsystems have high throughput, the main bottleneck for the performance is the network latency. Generally, the only method how to fight against the network latency is to limit the number of network roundtrips. In this section we describe performance optimization techniques adopted by Distributed OpenMI.

7.1. Direct Model Linkage

There are two special cases, where it would be very inefficient to adhere the linkage approach described in Section 4.6.

7.1.1. Server-side Model Linkage

If the accessing client links two remote models provided over same server, i.e. adds link between two *Remote Linkable Components* (RLC), by definition the linkage process would create *Model Provider* (MP) around both RLCs, create two new forwarders on the server for them and create RLC on both remote providing clients, so they can call each other (as described in Section 4.6). We can see that this is very inefficient since every call between inter-linked remote models must be transmitted over the accessing client.

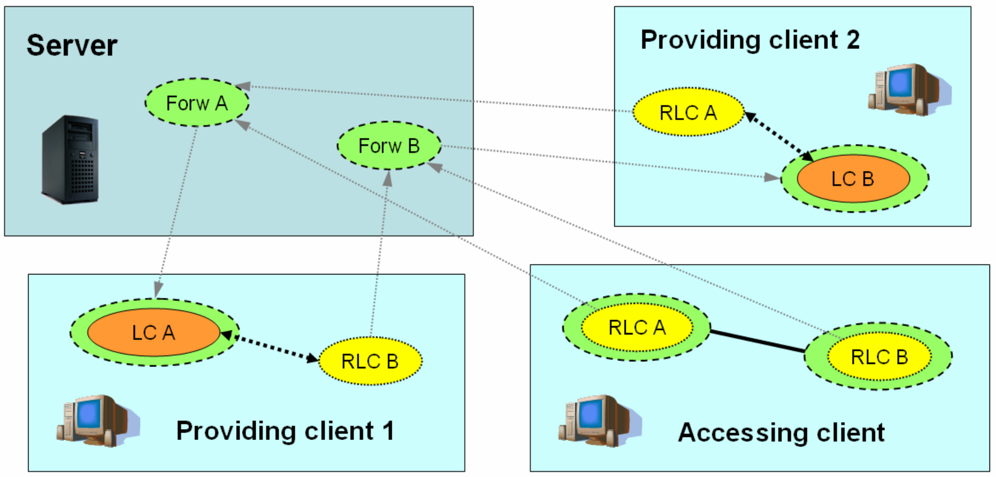


Figure 9 Server-side direct model linkage

As shown on Figure 9, Distributed OpenMI solves this special case differently. When link between two RLCs is added, Distributed OpenMI still creates MP over each RLC (necessary for case RLCs points to different servers), but when the server is going to create the forwarder for a particular MP, it looks first whether there is not already a forwarder for that model, and if so, reuses it. The effect is that calls between the models are now transmitted directly to the providing client.

7.1.2. Client-side Model Linkage

Similarly, when the link between two remote models provided by the same client (over the same server) is added, by default the Distributed OpenMI would create MPs over corresponding RLCs on the accessing client, create new forwarders for them on the server and create RLCs on the providing client. After that, all the communication between inter-linked models would be transmitted over the accessing client.

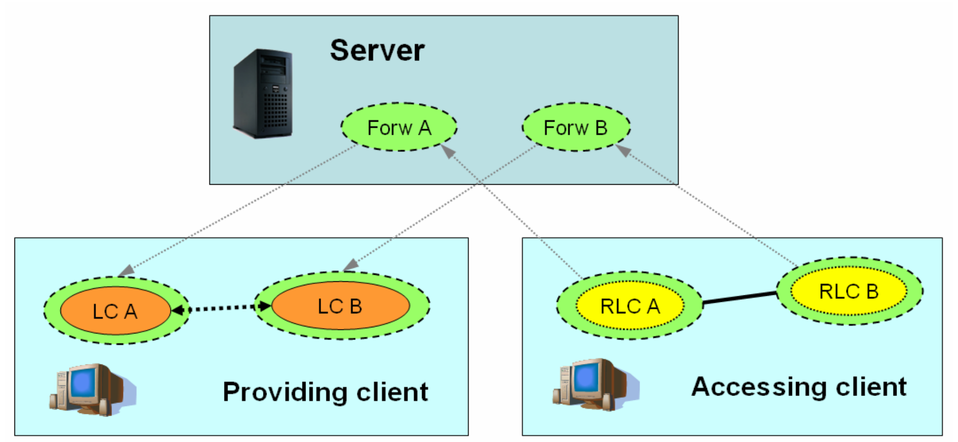


Figure 10 Client-side direct model linkage

Distributed OpenMI solves this special as follows. Before the client creates a RLC for some remote model, it first looks whether that RLC would not correspond to some local LC. If so, no RLC is created and the LC is used directly. This means that the models may now call their methods directly, without any assistance of Distributed OpenMI.

7.2. Caching and Piggybacking

The linkable component has several methods and properties that should not change the state of the component, according to the OpenMI Standard, and are used quite often. These methods are candidates for the caching and/or piggybacking. The caching may simply be implemented on the RLC level, the piggybacking necessitates the cooperation between RLC and MP, so that during one method call other values produced by LC are transferred to RLC and cached there.

Table 12 lists methods and properties of a linkable component and discusses the caching and piggybacking opportunities.

Method/property	**Comments**
Initialize	Called only once to initialize the LC.
ComponentID	These properties should not change the internal state of LC and should get the same value during whole lifetime of LC (after *Initialize* is called), thus it is desirable to piggyback them all together with a result of *Initialize* method or when one of the properties is read (for case the cache has been invalidated – explained below).
ComponentDescription	
ModelID	
ModelDescription	
TimeHorizon	
InputExchangeItemCount	Methods/properties should not change the internal state of LC. The exchange items are often read all at a time, so it is desirable to transfer them all together. On the other hand, iteration thru all exchange items may be expensive even if done locally. This naturally implies that the exchange items should be piggybacked and cached only if they are needed, i.e. when one of the method/properties is called and input/output handled separately.
GetInputExchangeItem	
OutputExchangeItemCount	
GetOutputExchangeItem	
AddLink	Methods change the internal state of LC.
RemoveLink	
Validate	Method should not change the state of LC and may be called multiple times, thus caching of a resulting value is desirable. Unfortunately, the validation can be expensive operation, thus piggybacking with other call is not possible. The cached value must be invalidated whenever the state of LC changes.
GetValues	Performs the computation step; changes the internal state of LC. However, the latest computed value may be cached, since it may be reused several times. The cached value must be invalidated whenever the state of LC further changes, because validation result may also change.
EarliestInputTime	Should not change the internal state of LC. Typically, the value of this property changes only after *GetValues* is called. This implies that piggybacking together with a result of *GetValues* will be useful. If the cached value has been invalidated, the value is read directly and cached thereafter.
Prepare	Methods change the internal state of LC, called maximum once during lifetime of LC.
Finish	
Dispose	
GetPublishedEventTypeCount	Should not change the internal state of LC. Published event types are

GetPublishedEventType	typically read all at a time, thus it is desirable to piggyback them all together when they are needed, i.e. when some of the methods is called.
Subscribe	Methods change the internal state of LC.
UnSubscribe	
SendEvent	
KeepCurrentState	Methods change the internal state of LC, called rarely.
RestoreState	
ClearState	
HasDiscreteTimes	Methods should not change the internal state of LC, and should return the same values during whole lifetime of LC. Typically, for a one combination of *Element Set* and *Quantity* all discrete times are read at a time. It is not possible to predict the requested combination of *Element Set* and *Quantity*, thus the piggybacking must be done for concrete combination first when one of these methods is called. Caching of the values for more than one combination is not useful, since typically the values are read only once during the computation.
GetDiscreteTimesCount	
GetDiscreteTime	

Table 12 Caching and piggybacking for linkable component methods and properties

The usage of caching and piggybacking may cause unexpected side effects in the case some method or property unexpectedly changes the internal state of the linkable component. To cope with that, Distributed OpenMI has the ability to adjust the caching or even turn it completely off. It is also for example possible to force Distributed OpenMI to invalidate the cache after state of LC is expected to change.

7.3. Intelligent Event Forwarding

As described in Section 4.7, the event receiver callback is associated with each RLC. In our integrating server model, the events produced by original LC are first send to a forwarder on the server, and then forwarded to all attached RLCs. The simple idea behind intelligent event forwarding is that not all RLCs do listen to a specific event types, thus some events do not need to be send to all RLCs. To support this behavior, the server must track which RLC called *ILinkableComponent*'s *Subscribe* and *UnSubscribe* method – the model provider forwarder is the right place where to do it.

Moreover, RLC sends the *Subscribe*/*UnSubscribe* calls only when it is really necessary. For example, if there are two subscribers for a same event type, it is sufficient to call *Subscribe* method only once, since RLC may broadcast the event to all subscribers locally.

One can see that events increase the number of network roundtrips in Distributed OpenMI heavily, thus in a production system it is desirable to limit the number of event subscriptions.

8. Deployment

The Distributed OpenMI system is divided into several software components. In this section we will describe these components, their installation and software prerequisites.

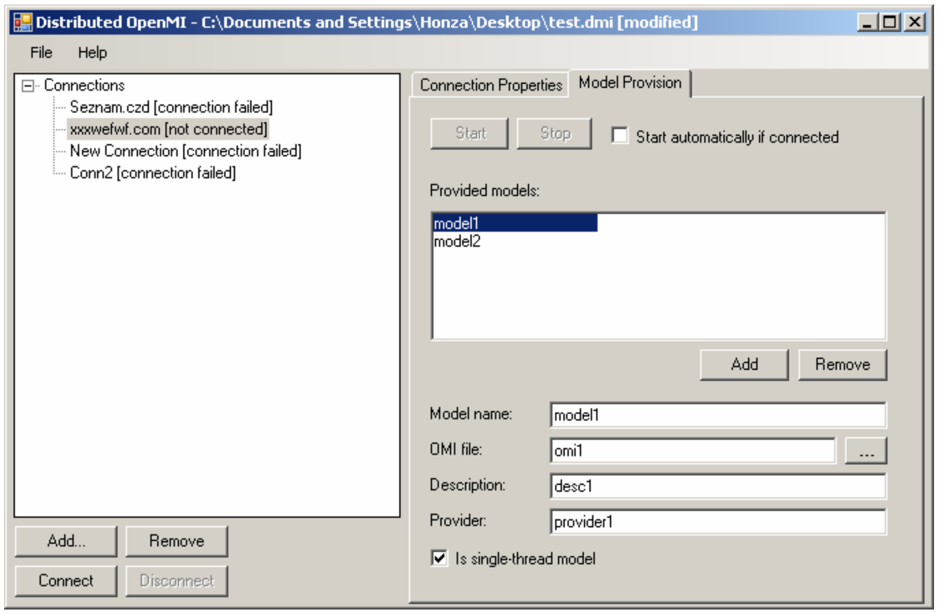
All components were developed in C# language and they are primary intended for the Microsoft Windows operating system with Microsoft .NET Framework 2.0 installed. The components may possibly run on other operating systems, using MONO framework (developed in [Mono]) or DotGNU (developed in [DotGnu]), because our implementation is generally platform independent. However at the time of the development (Q1/2007), the support for .NET 2.0 and necessary libraries has been fully implemented neither by MONO nor by DotGNU. These frameworks are currently the only opportunity how to run the existing system on Linux. This implies that only Microsoft Windows is currently "officially" supported by Distributed OpenMI. After MONO finishes the necessary support for .NET 2.0 (planned to Q3/2007), there is nothing in the way to migrate our system to Linux and potentially other operating systems.

8.1. Client

Both providing and accessing clients are encapsulated in the *DHI.OpenMI.Distributed.Client.exe* assembly. To use this assembly, the OpenMI Standard software must be installed on the local computer.

To start the accessing client, the type implementing RLC (*DHI.OpenMI.Distributed.Client.RemoteLinkableComponent*, contained in the client assembly) must be explicitly instantiated and RLC thereafter initialized using *ILinkableComponent*'s *Initialize* method – this is typically done via the OMI file using the standard OpenMI deployment software.

To start the client front-end application, simply run the *DHI.OpenMI.Distributed.Client.exe* assembly. The application snapshot is depicted on Figure 11.



Connections
- Seznam.czd [connection failed]
- xxxwefwf.com [not connected]
- New Connection [connection failed]
- Conn2 [connection failed]

☐ Start automatically if connected

Provided models:
- model1
- model2

Model name:	model1
OMI file:	omi1
Description:	desc1
Provider:	provider1

☑ Is single-thread model

Figure 11 Distributed OpenMI client application

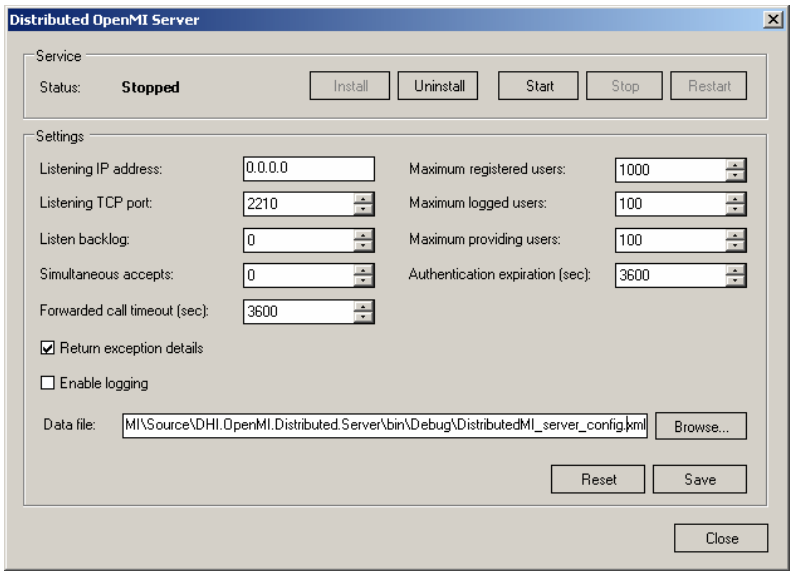
Using the client application, the user is able to:

- Manage the connections to Distributed OpenMI servers
- Browse the providing clients connected to the server
- Browse the provided models
- Display the properties of a provided model
- Generate the OMI file to access a remote model
- Provide own local models

When a user provides some models, the client application must keep running and the connection to server must remain open. All settings done in the client application may be saved to a XML file (by default, it has *DMI* extension), and reused later. To automatically open such file, we can add path to it as a single command-line argument when starting the client application. This may be useful for example to automatically start the providing client on a computer startup.

8.2. Server

The server is encapsulated in *DHI.OpenMI.Distributed.Server.exe* assembly. It may either run as standalone application or may be installed as Windows Service. For the installation of the service and configuration of the server, special tool has been developed.


Distributed OpenMI Server

Service

Status: **Stopped**

Settings

Listening IP address: 0.0.0.0

Listening TCP port: 2210

Listen backlog: 0

Simultaneous accepts: 0

Forwarded call timeout (sec): 3600

Maximum registered users: 1000

Maximum logged users: 100

Maximum providing users: 100

Authentication expiration (sec): 3600

☑ Return exception details

☐ Enable logging

Data file: MI\Source\DHI.OpenMI.Distributed.Server\bin\Debug\DistributedMI_server_config.xml

Figure 12 Distributed OpenMI server configuration tool

To run this tool, simply run *DHI.OpenMI.Distributed.Server.exe* with no command-line options. The */help* command line option will display help dialog describing other start-up options. Note that both server and the configuration tool do not need the OpenMI Standard to be installed on the computer.

8.3. Tester

For testing of Distributed OpenMI the *DHI.OpenMI.Distributed.Tester.exe* application has been developed. This application is able to prepare an OpenMI composition of remote models provided by various clients over various servers. The exact definition of the composition is supplied using command-line arguments. The application performs following tasks:

- Automatically start the requested server processes
- Automatically start the providing client processes, and instruct them to connect to corresponding servers and provide the requested models
- Generate OMI files for remote models
- Create the standard OpenMI composition consisting of both remote and local models, add a trigger and link models appropriately
- Save the OpenMI composition to OPR file, so it may be opened by *OmiEd* (a standard OpenMI deployment application)
- Run the composition, if requested

All remote and local models taking part in the composition are special linkable components (*DHI.OpenMI.Distributed.Tester.TestLinkableComponent*), which reside in the *DHI.OpenMI.Distributed.Tester.exe* assembly. These LCs tests various aspects of Distributed OpenMI, for example:

- Check LC methods are called from same thread
- Check LC methods are called in correct order
- Check client-side direct model linkage
- Test event sub-system

To run the tester application, the OpenMI Standard software must be installed on the computer.

8.4. User Documentation

Because the Distributed OpenMI system is currently a prototype, there is still no user documentation available. However, it will be delivered in the future, and will mainly focus on:

- Installation and usage of the client application
- Installation and configuration of the server
- Optimization of performance
- Solution of possible problems (caused e.g. by caching mechanism)

9. Conclusion

The Distributed OpenMI system has shown that it is possible to seamlessly distribute the OpenMI composition of any OpenMI-compliant models to different computers. Distributed OpenMI is a system, which is simple to configure and to use, and thus may be adopted by users without advanced computer skills. Current system is reliable in the way that it may be used for real-world projects.

Moreover, we proved that it is possible to spread single-thread call-stack to several computers, even in the integrating server environment, whilst ensuring that nested calls are executed on a same thread as the causal calls. This feature is necessary to guarantee high performance thanks to low thread utilization and is of a vital importance for various software components. The classical distributed computing philosophy is "distribute the data to the computers, and let each one to compute locally", we say "leave the data locally, and distribute the computation". Thanks to the independence of YA-RPC protocol, our novel approach may be reused by other software systems.

However, current Distributed OpenMI system does not incorporate all the functionality proposed in the specification of this thesis. The following list summarizes in points all parts of the specification and discusses how they are realized in current Distributed OpenMI:

- **System capable of linking OpenMI models across computers**
  Completely fulfilled
- **Hub servers for Windows and Linux**
  As described in Section 8, the server software runs on Microsoft Windows using Microsoft .NET Framework platform. .NET Framework 2.0 is currently completely supported neither by MONO framework nor DotGNU, what is the presumption to run the existing server on Linux. The Java implementation of the server would necessitate providing Java implementation of the YA-RPC protocol, which was beyond the scope of this thesis. In addition, this might be a useless work because the release of MONO 2.0 is planned to Q3/2007.
- **The clients providing local models**
  Completely fulfilled
- **Transparent access to remote models for legacy OpenMI software**
  Completely fulfilled

- **Clients for .NET and Java**

  As described in Section 8, the client software runs on Microsoft Windows using Microsoft .NET Framework. The Java client has not been implemented, because OpenMI Association stopped the support for Java in the OpenMI Standard. After that, there was no reason why to implement a Java client.
- **Strong encryption, privileges management (optionally)**

  These features are not implemented, because they are unnecessary for the current non-commercial release of Distributed OpenMI. Even if they would be implemented, an additional work is still needed to suit the needs of a commercial environment – Section 10 gives more details about that. This led to a decision to skip the implementation of these features.
- **Transparent communication failover (optionally)**

  The natural place for integration of the communication failover is the RPC protocol layer. Because during the development of the system the necessity to implement a proprietary communication protocol arisen (as described in Section 6), we would need to implement the failover capability there. The amount of work for that would be enormous, and thus has been skipped. However, YA-RPC may be extended with this feature later.
- **Recommendations for true parallel computing (optionally)**

  The recommendations for an asynchronous computation have not been made because the OpenMI Association does not give an indication that such functionality will ever be adopted by OpenMI.
- **Reliable system usable in real situations**

  Completely fulfilled
- **Demonstration on a real scenario**

  The Distributed OpenMI system has been tested on real setups of *MIKE 11*, *MOUSE* and *EPANET* modeling software (provided by *DHI Water, Environment and Health*). Unfortunately, both the modeling software and setups are proprietary software, which could not be attached to the thesis. However, the statement of DHI, a.s., the company who carried out the tests, is attached to the thesis in a separate paper.

All decisions to not implement specific proposed features have been consulted with the supervisor of the thesis.

10. Future Work

Although Distributed OpenMI delivers a system usable in real-world projects, there are still some features, which need to be implemented before the commercial expansion of the software, i.e. before the software may be shipped as a boxed product. These features are:

- **Privileges**
  Privileges enable the users to specify, which clients may access their models, whether additional arguments may be supplied to initialization of a linkable component, whether users may see exceptions generated by particular model…etc.
- **Client administration tool on server**
  The tool for administration of clients on the server, which would be used to manually control the registration (e.g. based on a received payment), remove the server clients…
- **Encryption**
  In a production environment, the model data are often not public, or even may be confidential (e.g. simulations for military purposes). Since the data may be transferred over unsecured networks, potential attacker may sniff the communication between the clients and the server. The only sufficient protection is the encryption of the communication, most likely implemented on the YA-RPC protocol layer.
- **Transparent network failover**
  Internet consists of networks which differ in the quality a lot. TCP is connection oriented protocol, what unfortunately means that if the network does not transmit TCP packets for a period of time, the TCP connection may be lost. If the connection between any client and corresponding server participating in the Distributed OpenMI composition is lost, even for small period of time, the whole simulation fails. The solution is to implement a failover mechanism, most likely on the YA-RPC level, which would simply try to reconnect to the server for a specific period of time in case TCP connection has been accidentally lost. If the re-connection succeeds, the state of the remote calls must also be restored and lost remote calls must be re-sent.

- **Java client in case OpenMI Association decides to support Java again**

  Java platform may be supported by OpenMI in the future again, what may request the creation of a Java client. It may be possible to re-compile existing .NET assemblies to Java byte-code automatically without much effort (e.g. using IKVM.NET framework, [Ikvm]).

11. References

[Bir83]	A.D. Birrell, B.J Nelson (1983): Implementing Remote Procedure Calls, XEROX CSL-83-7.
[Dce97]	The Open Group (1997): DCE 1.1: Remote Procedure Call
[Dcom98]	The Open Group (1998): The COM/DCOM Reference, Documentation for ActiveX Core Technology
[DotGnu]	DotGNU Project, http://www.gnu.org/software/dotgnu/
[Ecm06]	ECMA International (2006): Standard ECMA-335, Common Language Infrastructure (CLI)
[Fast]	The FastRPC protocol documentation, http://fastrpc.sourceforge.net/
[Omi05]	Peter J.A. Gijsbers, R. Brinkman, J.B. Gregersen, S. Hummel, S.J.P. Westen and others (2005): The org.OpenMI.Standard interface specification.
[Ikvm]	IKVM.NET Project, http://www.ikvm.net/
[Java03]	Sun Microsystems, Inc. (2003): JavaTM 2 Platform, Standard Edition, v1.4.2, API Specification
[Jon02]	A. Jones, J. Ohlund (2002): Network Programming for Microsoft Windows, Second Edition
[Mono]	The MONO Project, http://www.mono-project.com/
[Net07]	Microsoft Corporation (2007): .NET Framework Reference
[Omg04]	Object Management Group, Inc. (2004): Common Object Request Broker Architecture: Core Specification
[Rfc793]	Information Sciences Institute, University of Southern California (1981): RFC 793 Transmission Control Protocol
[Rfc1831]	R. Srinivasan (1995): RPC: Remote Procedure Call Protocol Specification Version 2
[Rfc3022]	P. Srisuresh, K. Egevang (2001): RFC 3022 Traditional IP Network Address Translator (Traditional NAT)
[Rfc4122]	P. Leach, M. Mealling, R. Salz (2005): RFC 4122 A Universally Unique IDentifier (UUID) URN Namespace
[Xml99]	Dave Winer (1999): XML-RPC Specification
[Web04]	D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard (2004): Web Services Architecture, W3C Working Group Note
[Win07]	Microsoft Corporation (2007): Windows API Reference