Performance Analysis of Persistence Technologies for Cloud Repositories of Models

Juan Pablo SalazarÁlvarez[1], Elena Gómez Martínez[1], and Miguel de Miguel[1,2]

[1] Center for Open Middlewarre, Universidad Politécnica de Madrid (UPM), Pozuelo de Alarcón - Madrid, Spain
[2] DIT, Universidad Politécnica de Madrid (UPM), Madrid, Spain

Abstract. The growing adoption of Model Driven Development (MDD) in companies during last decade arises some model interchange problems. Companies need support to interchange models and reuse parts of them for developing new projects. Traditional tools for model edition and model interchange have different performance issues related to the models storage. There are mainly two styles to organize the persistence of models into repositories: a complex and large model or a large amount of small models. This last approach is common in companies that generate software from models. In this paper, we analyse performance properties of different persistence technologies to store small/medium-scale models, the analysis results should be considered in the design of model repositories in the cloud. With this aim, we have designed and developed a generic architecture to evaluate each persistence technology under similar situations.

1 Introduction

During the last decade, Model Driven Development (MDD) has been acquired by companies to improve quality, productivity and the reuse of their software development. Models are not only an abstraction of the system under development, but also the system is generated from them. This reuse process is especially important in MDD approaches that generate automatically software code from models. Besides, multinational companies build development teams that are geographically dispersed. MDD developers require virtual model interchange support, which, currently, is well supported for programming languages, but not yet for models. Tools for managing and sharing these models are needed in order to reuse models in a collaborative industrial development ecosystem. Collaboration is usually supported by model interchange and versioning supported in a model repository. Therefore, apart from version control, querying and transformation, one of the key functionality of these tools is the persistence.

In this paper, a model repository in the cloud provides the storage of models with the possibility of accessing and updating such models [3]. The model repository supports any modeling language, even a coexistence of them. To organize models persistence, there are chiefly two styles: either a unique very complex or large model that contains all the elements or a great number of small models with partial functionality. But also there is a lot in between, a particular example is model fragmentation for large models; in [25], large models can be split and persisted in fragments as small-scale models. There are several works in the literature [3, 4, 10, 11, 14] that evaluate repositories for large-scale models. Nevertheless we focus on the persistence technologies used by those companies that have developed lots of small/medium-scale models. This approach is common in companies that generates software from models. At these companies, the amount of models and their relationships (cross-references) are becoming a problem to be concerned with. Furthermore, maintenance systems are also a weak point in these companies, due to the amount of reused elements among models.

There are two factors that make very complex the construction of applications into a large-scale model: the size of generated software from models (more than 20 times the size of model) and the model accuracy level (because models are the inputs for the construction of executable applications). At this situation, developers tend to decompose their applications into small/medium-scale models (size less than 1M). Santander Group[3] is an example of company that have been using MDD approach for the construction of banking web applications during last decade. Several thousand engineers in different locations use domain specific languages (DSL) for developing software applications. The medium size of these DSL-based models is less than 100K. But each application includes hundreds of models.

There are different alternatives to persist models. By default, modeling tools store models in a standard file approach. They tipically uses XML/XMI for interchange aim. However, other persistence layers have emerged in last years, such as relational databases and NoSQL databases. The relational databases provide mechanisms to index and access objects with SQL queries. The NoSQL databases are based on several *schemaless* database paradigms, among them document or graph.

The ultimate aim of our research is to achieve a framework to share cloud-based small/medium-scale models developed collaboratively whichever modeling language they use. As a first step, we analyse different persistence technologies for performance perspective in this work. For this purpose, we have designed a generic architecture in order to abstract each persistence layer details. The architecture also abstracts the modeling language details. Moreover we have implemented five persistence backends: two file-based solutions and three databases (a relational, a document-based and a graph-based one).

Concerning performance evaluation, we have executed a case study with two usage scenarios to analyse them by means of testing techniques. As performance metrics, we have followed those proposed in Software Performance Engineering (SPE) [26]. Thus, we focus on response time, scalability and storage cost.

The rest of the paper is organized as follows. Firstly, Section 2 outlines the basic concepts and state of the art. Section 3 describes a case study which is analysed in Section 4. Finally, Section 5 states some conclusions.

[3] http://www.santander.com

2 Background and State of the Art

This section defines some concepts related to Model-Driven Development, modeling languages and persistence technologies that will be used in the remainder of the paper. We also outline some collaborative MDD tools.

MDD aims to overcome the complexity of systems and their development by abstracting relevant information, by working at the model instead of the code level. In MDD, models specify the structure, behaviour and requirements of a system. A metamodel provides the syntax specification of a language; it describes the concepts and relationships of a certain domain [10]. Thus, a model is an instance of its corresponding metamodel. Some modeling languages are Business Process Model and Notation (BPMN) [21], the Unified Modeling Language (UML) [20], Meta-Object Facility (MOF) [22] and Ecore [28], among others.

Eclipse Modeling Framework (EMF) [28] is a modeling framework, integrated in Eclipse, for building tools and other applications based on a structured metamodel. EMF facilitates the definition and instantiation of metamodels and runtime support for the models including change notification, persistence implementation and manipulation of EMF objects.

In some sections we use UML and MARTE (Modeling and Analysis of Real-Time and Embedded systems) profile [19] to provide performance information of our benchmarks. MARTE includes the sub-profile Generic Quantitative Analysis Model (GQAM) designed for performance analysis purposes.

2.1 Model Persistence Technologies

There are three chief approaches to store models and metamodels: XML/XMI files, relational databases through object relational mappings and model repositories. In the following, we outline some of them.

Models are typically stored using XML-based formats, like XMI (XML Metadata Interchange) [23], which have to be parsed to build in-memory models. XML parsers are in charge of accessing to its data content and modifying its content or its structure. Parsers carry out two main processes: serialization and deserialization. Serialization process converts an in-memory object into an XML stream. Reversely, deserialization converts XML streams in wire-format objects in memory.

EMF defines the *Resource* interface to represent a physical storage location [28]. The EMF *Resource* provides four basic operations involved in moving models between memory and persistence: load, save, update and delete. *XMLResource* is the interface created by EMF capable for serializing any model into a XML file. In a similar manner, *XMIResource*, inherited from *XMLResource*, is used to serialize any model using XMI 2.0. This is the default serialization adopted by EMF. EMF also provides a simple serialization of the models through *BinaryResource*, which saves the models in binary files.

Teneo [29] is a model-relational mapping and runtime database persistence solution for EMF. Teneo integrates Hibernate and EclipseLink. So, it derives a relational schema and an object-oriented API from an Ecore metamodel. Relational databases provide mechanisms to index and access objects via SQL queries. Teneo can use MySQL [24], among others, as target database.

Concerning model repositories, Connected Data Objects (CDO) is a *de facto* standard solution to handle models and metamodels in EMF. CDO is both a development-time model repository and a runtime persistence framework [30]. CDO allows models to be persisted into all kinds of database backends like major relational databases or NoSQL databases. However, in practice, only relational databases are commonly used.

Morsa [10,11] is a prototype of a model repository of large scale EMF models based on a non relational database. Morsa uses a document-oriented database as persistence backend, specifically MongoDB [2, 16]. It therefore stores EMF models as collections of documents. The work of Espinazo-Pagán et al. [10, 11] also compares Morsa with CDO and XMI files. As shown, Morsa exhibits better performance than other approaches for large-scale models.

Neo4EMF is a model repository built on the top of the NoSQL graph-based database Neo4j [17]. In Neo4EMF, EMF-based models can be described in terms of graphs concepts, since there is an immediate mapping between the two representations, described in [4]. As the aforementioned model repositories, Neo4EMF aims to scale large scenarios. In [4], a comparison between EMF standards parameters (XMI files), CDO and Neo4EMF is done. The experiments are performed over the Java MoDisco Metamodel [31] and focused on large-scale scenarios. Their results show that Neo4EMF outperforms other alternatives for this kind of scenarios.

EMFStore [14] is a operation-based framework for versioning EMF models, that includes a model repository. Therefore, unlike the above approaches, EMFStore supports model evolutions. Internally, EMFStore handles XMI files that envelopes internal models for operations and changes. EMFStore lacks a mechanism to solve cross-document references between models that have been developed with any other tool.

In this paper, our main objective is the evaluation of persistence mechanisms for large amounts of small-scale models. We have then analysed some of the persistence technologies outlined in this section. Specifically, we have evaluated:

– File-based mechanisms: By means of EMF *Resource*, we have developed two model repositories. The first one uses XMI (*XMIResource*). The second one is a model repository with binary files (*BinaryResource*).
– Relational database-based mechanisms: Since Teneo connector is more simplified than CDO, we have studied Teneo with MySQL database.
– Non-relational databases mechanisms: We have analysed the document-based Morsa and the graph-based Neo4EMF.

We have not evaluated EMFStore, since it was not sufficient mature when implementation phase was carried out.

2.2 Collaborative MDD tools

Insofar our ultimate aim of our research is to achieve an efficient tool for collaborative model-driven development, we briefly outline some of these tools. Although, our aim does not deal with editing, many of these tools also offers a model editor tool among their functionalities.

ModelBus [12] is a tool integration platform for addressing the functional connection of services provided by different modeling tools, such as editing or transformation among others [5]. It is based on a virtual bus-like service-oriented architecture. It allows therefore the data exchange between models in a collaborative environment. As pointed out in [10], the ModelBus repository is a web service application that manages an embedded Subversion engine which implements the actual repository. However, it does not allow partial access to models.

Modelio is an enterprise-level open source modeling solution to develop UML-based models [27]. It supports requests, extraction and modification of models, which can be accessed through a Java API. Modelio uses Teamwork Manager tool to endow collaborative development environment [7]. This module provides a model repository with a central storage area, which is locally replicated by each user. It internally integrates a Subversion repository [1] in order to share modeling projects. Unlike our proposal, EMF is not supported by Modelio [7].

MagicDraw [18] is a commercial modeling framework based on UML standard. By means of Teamwork Server, it provides a central repository for storing models. Models can be accessed, reviewed or modified. Internally, Teamwork Server uses an internal proprietary code to implement a native versioned file-based repository. MagicDraw supports EMF through a conversion into UML.

MetaEdit+ [15] is a commercial domain-specific modeling environment composed of two products: MetaEdit+ Workbench for creating modeling tools and generators and MetaEdit+ Modeler for editing models with multiple users, projects and platforms. It also provides a repository for storing models.

Obeo Designer [9] is a commercial model-driven approach to specify models. It is based on EMF and plugged into Eclipse Platform IDE. Obeo Designer relies on a central CDO repository to facilitate collaborative work [6].

GenMyModel [8] is a commercial modeling platform. It allows to diagram models through web editors among other functionalities, such as to provide a model repository. It focusses on UML, although other languages are available.

3 Case Study: A Generic Architecture

In order to compare the target persistence technologies, a software architecture has been designed for a generic repository. In the following, this architecture and its usage scenarios are described.

3.1 Architectural Design

We have developed a software architecture for a generic repository. The purpose of this architecture is to abstract any modeling language, as well as any persistence solution. The architecture plays with abstract elements to define the same characteristics and behaviours for different implementations. Thus, the obtained results can be compared without benefit or harm to any technology.

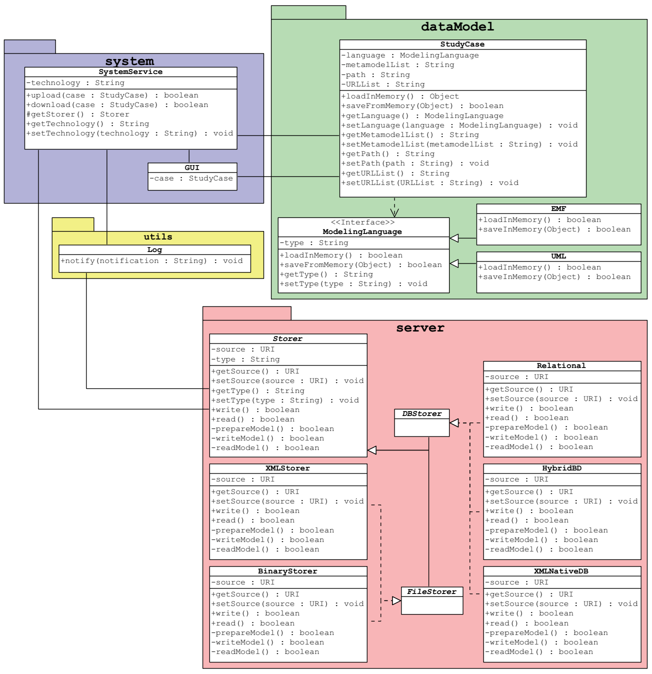

system
SystemService
-technology : String
+upload(case : StudyCase) : boolean
+download(case : StudyCase) : boolean
#getStorer() : Storer
+getTechnology() : String
+setTechnology(technology : String) : void
GUI
-case : StudyCase
dataModel
StudyCase
-language : ModelingLanguage
-metamodelList : String
-path : String
-URLList : String
+loadInMemory() : Object
+saveFromMemory(Object) : boolean
+getLanguage() : ModelingLanguage
+setLanguage(language : ModelingLanguage) : void
+getMetamodelList() : String
+setMetamodelList(metamodelList : String) : void
+getPath() : String
+setPath(path : String) : void
+getURLList() : String
+setURLList(URLList : String) : void
<<Interface>>
ModelingLanguage
-type : String
+loadInMemory() : boolean
+saveFromMemory(Object) : boolean
+getType() : String
+setType(type : String) : void
EMF
+loadInMemory() : boolean
+saveInMemory(Object) : boolean
UML
+loadInMemory() : boolean
+saveInMemory(Object) : boolean
utils
Log
+notify(notification : String) : void
server
Storer
-source : URI
-type : String
+getSource() : URI
+setSource(source : URI) : void
+getType() : String
+setType(type : String) : void
+write() : boolean
+read() : boolean
-prepareModel() : boolean
-writeModel() : boolean
-readModel() : boolean
DBStorer
FileStorer
Relational
HybridBD
XMLNativeDB
XMLStorer
BinaryStorer
-source : URI
+getSource() : URI
+setSource(source : URI) : void
+write() : boolean
+read() : boolean
-prepareModel() : boolean
-writeModel() : boolean
-readModel() : boolean

Fig. 1. Class Diagram of the architecture for any repository abstracting persistence.

The generic architecture is depicted in the UML Class Diagram in Figure 1. Its modules are:

`system`, which contains the graphical user interface (GUI) and the *SystemService* that can be accessed through the GUI. The technology is also selected in the *SystemService*.

`dataModel`, whose main class is *StudyCase* which represents the models sets, as well as the modeling language that they support. Thanks to the *interface ModelingLanguage* is possible to unify the behaviour of each language.

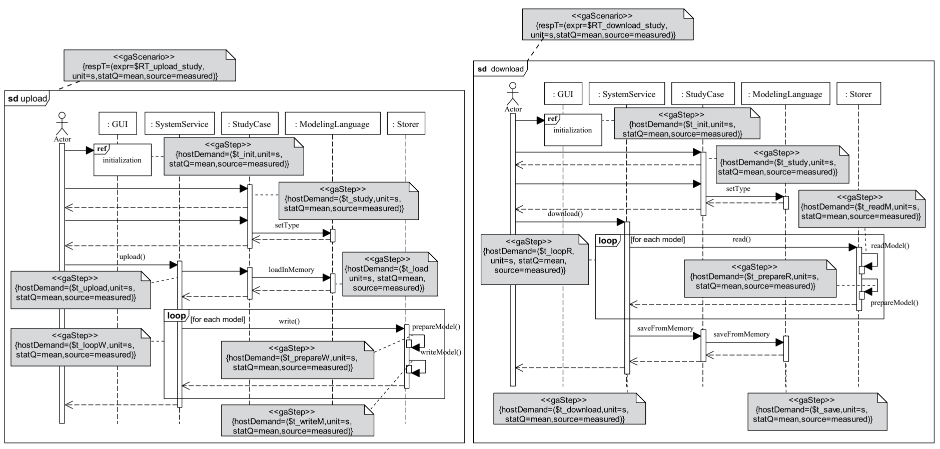
`server`, that contains the *Storer* and their corresponding implementations considered in our analysis, one per each persistence technology. Each storer implements these main functions: *readModel* (recover the model), *writeModel* (persist the model) and *prepareModel* (adapt the model to persistence technology). Eventually, the differences between storers were reduced.

`utils`, that contains the log for obtaining the metrics.

3.2 Usage Scenarios

According to Smith and Williams [26], performance scenarios are those Use Cases of a software system that are executed frequently or are critical to the user's perception of performance. To compare each technology, we have selected two main basic usage scenarios of the generic repository as storage location: `Upload` and `Download` scenarios. Firstly, a model developer is able to upload to the generic repository a large number of small-scale models (`Upload`); secondly, a model developer is able to download from the generic repository a large number of small-scale models (`Download`).

Figure 2(a) depicts the UML Sequence Diagram[4] describing the `Upload` scenario. This scenario involves the following steps: i) the user selects the persistence technology and, consequently, ii) the system initializes it taking into account the targeted modeling language; iii) it thus reads the XMI files containing the models, iv) parses them and loads them in memory as EMF *eObjects*; v) it then writes each model to the specific persistence technology in sequence. Previously, each model must be prepared (or processed) according to the specific storer.



<<gaScenario>> {respT=(expr=$RT_upload_study, unit=s,statQ=mean,source=measured)}
sd upload
Actor
: GUI
: SystemService
: StudyCase
: ModelingLanguage
: Storer
ref
initialization
<<gaStep>> {hostDemand=($t_init,unit=s, statQ=mean,source=measured)}
<<gaStep>> {hostDemand=($t_study,unit=s, statQ=mean,source=measured)}
setType
upload()
loadInMemory
<<gaStep>> {hostDemand=($t_load, unit=s, statQ=mean, source=measured)}
<<gaStep>> {hostDemand=($t_upload,unit=s, statQ=mean,source=measured)}
loop
[for each model]
write()
prepareModel()
writeModel()
<<gaStep>> {hostDemand=($t_loopW,unit=s, statQ=mean,source=measured)}
<<gaStep>> {hostDemand=($t_prepareW,unit=s, statQ=mean,source=measured)}
<<gaStep>> {hostDemand=($t_writeM,unit=s, statQ=mean,source=measured)}
<<gaScenario>> {respT=(expr=$RT_download_study, unit=s,statQ=mean,source=measured)}
sd download
Actor
: GUI
: SystemService
: StudyCase
: ModelingLanguage
: Storer
ref
initialization
<<gaStep>> {hostDemand=($t_init,unit=s, statQ=mean,source=measured)}
<<gaStep>> {hostDemand=($t_study,unit=s, statQ=mean,source=measured)}
setType
<<gaStep>> {hostDemand=($t_readM,unit=s, statQ=mean,source=measured)}
download()
<<gaStep>> {hostDemand=($t_loopR, unit=s, statQ=mean, source=measured)}
loop
[for each model]
read()
readModel()
<<gaStep>> {hostDemand=($t_prepareR,unit=s, statQ=mean,source=measured)}
prepareModel()
saveFromMemory
saveFromMemory
<<gaStep>> {hostDemand=($t_download,unit=s, statQ=mean,source=measured)}
<<gaStep>> {hostDemand=($t_save,unit=s, statQ=mean,source=measured)}

(a) `Upload` (b) `Download`

Fig. 2. UML Sequence Diagram describing the `Upload` and `Download` scenarios.

We use the `Download` scenario for the load performance in persistence technologies; the scenario is depicted in the UML Sequence Diagram (SD) of Figure 2(b). This scenario consists of the following steps: i) the initialization of the persistence technology by the user selection; ii) the user chooses the modeling language to recover the stored models. iii) Each model is then read from the specific persistence technology in sequence, iv) recovered and load in memory as an EMF *eObject*; v)then they are copied to XMI files according to the above selected modeling language.

In both scenarios, during step iv) the models are fully load and resolved in order to be able to allow the system to exchange the persistence technology, from the XMI file to the persistence technology selected and vice versa.

As observed, Figures 2(a) and 2(b) are annotated with performance information according to the MARTE profile [19]. These annotations indicate the actions duration collected by experimental tests, as detailed in Section 4.

[4] The reader should note that we have added some grey notes (SD) in the UML diagrams. They are performance annotations that will be briefly outlined in Section 4.

4 Empirical Evaluation

This section compares the selected persistence technologies from software performance viewpoint. For this purpose, the execution environment of the experiments are firstly determined. Then, we define those performance metrics to be evaluated, as well as the experiments results.

4.1 Execution Environment

At this point, we will present results in this section based on two well-known metamodels for benchmarking the selected persistence technologies: Extended Purchase Order (EPO), a metamodel proposed in [28] based on the well-known example from XML Schema Part 0: Primer Second Edition[5]; and Extended Library (LIB), a metamodel taken from an Eclipse tutorial[6]. The two metamodels cover as many features and data types from those available in EMF. In this article we use these metamodels for benchmarking purposes, but we have done similar evaluations and obtained similar results with some other languages such as the *Notation* metamodel that uses Graphical Modeling Framework (GMF) in Eclipse to represent diagrams concepts. We like to emphasize that the metamodels characteristics such as depth of containment tree, number of cross references are irrelevant in the approach of these experiments since the repository is oriented to store models from modeling languages of any kind.

The model instances (models) are automatically generated by Generators which exploits all the attributes and relationships in the metamodels. We have generated two default testbed with 2000 models of EPO and LIB for each one. The number of elements per model are 211 and 313, respectively; the model size is 30 KBytes in EPO metamodel and 47 KBytes in LIB metamodel.

The generic repository has been deployed in a laptop computer running Windows 7 Professional (64 bits) operating system. The computer system characteristics are Intel Core i7 processor 3720QM(2.60GHz) with 16 GB RAM of DDR3 SRAM(1866MHz). Experiments have been executed on Eclipse Helios via Java Application (when it was possible, if not, an Eclipse Application) running Java SE Runtime Environment version 1.7. The Java Virtual Machine (JVM) has been restarted for each measure as well as for each repetition of each measure.

Concerning each specific persistence technology, for all the technologies the default configuration has been used, whenever it was possible and barring error. In particular, we have implemented Teneo/Hibernate with a relational database, MySQL Workbench Community (GPL) for Windows version 6.1.4. As NoSQL databases, we have tested a document-oriented database MongoBD through Morsa [10,11] version 1.0.0 and a graph database named Neo4j via Neo4EMF [4] version 0.1 (November 2013).

[5] http://www.w3.org/TR/smlschema-0/
[6] http://help.eclipse.org/luna/nav/21_1

4.2 Performance Results

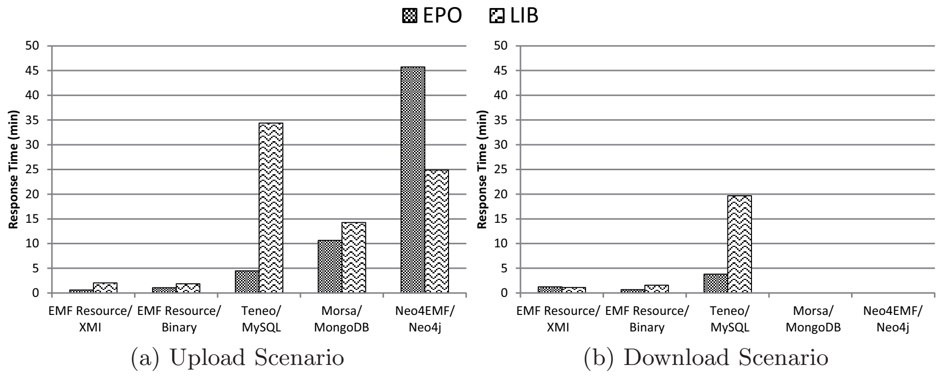
We have compared each persistence technology by considering the following performance properties proposed in [26]: response time, scalability and storage cost. Although there are others, we focus on those critical ones for our aim, a collaborative framework development to share models. In the following, we describe our experiments and the obtained performance metrics in order to analyse them.

Response time *Response time* is the time interval between a user request of a service and the response time of the system [13]. In this paper, the response time can be defined as the scenario duration for each persistence technology.

Performance information concerning atomic actions and scenario durations has been collected in experimental tests. According to MARTE profile, atomic actions are represented by `<<GaStep>>` stereotype, where `hostDemand` tag specifies its corresponding average measured execution time; and scenario durations are specified using `<<GaScenario>>` annotation, as illustrated in Figure 2. Seconds are the measurement unit and mean is the statistical measure.

Figures 3 (a) and (b) illustrate the average response time of `Upload` and `Download` scenarios for each persistence technology. We have analysed the selected persistence technologies considering these situations: i) how it performs depending on the model size; and, ii) how it performs each scenario using the same metamodel. We have obtained these results with the two default testbeds.

As observed in Figures 3 (a) and (b), file-based persistence solutions perform better than database-based ones for both scenarios. Focusing on `Upload` scenario in Figure 3 (a), EMF Resource/XMI outperforms the rest of persistence technologies. Specifically, `Upload` scenario EMF Resource/XMI spends 38.35 seconds and 65.80 seconds for EPO and LIB respectively. As observed, the response time of Teneo/MySQl is the most affected by model sizes, since it is multiplied by 7.5 with LIB models. Remark the values obtained with Neo4EMF/Neo4j, in spite of EPO model size is smaller than LIB one, its `Upload` scenario with LIB models performs better. We guess that this response time is due to Neo4EMF resolves the interrelations within models during the `Upload` scenario.



(a) Upload Scenario (b) Download Scenario

Fig. 3. Response time for each persistence technology by injecting default testbeds.

Concerning `Download` scenario depicted in Figure 3 (b), file-based persistence mechanisms are the fastest solutions. In the case of EMF Resource/Binary, the response time is proportional to the model size. However, response time is not affected by the size in EMF Resource/XMI. As in the previous scenario, Teneo/MySQL performs poorly for LIB models. Furthermore, the response time is almost multiplied by 10 comparing with EPO and LIB models. Note that Morsa/MongoDB has not been performed since prototype could not be run successfully. In the case of Neo4EMF/Neo4j, it was not possible to recover a full load of a model from the repository, so the measures had to be dismissed.

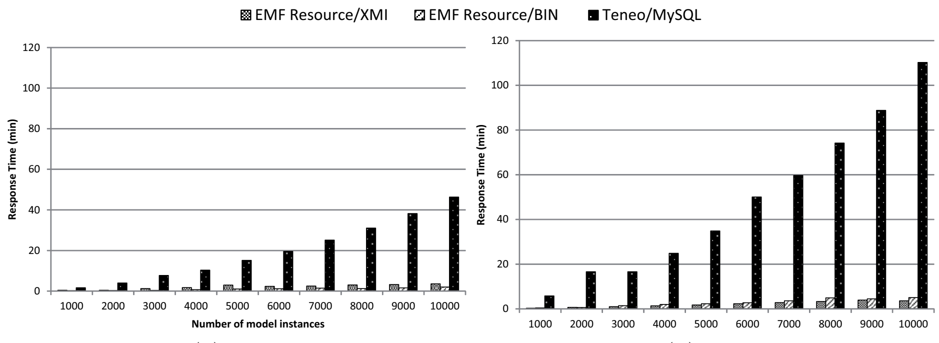
If the same persistence technology are analysed comparing the results in the two scenarios, we observe that `Upload` scenario requires more time than `Download` scenario in file-based mechanisms. An exception is Teneo/MySQL that spends similar response time for both scenarios.

From the response time point of view, we observe that the best persistence solutions (for a very populated model repository of small-scale models) are those based on files. Moreover, it would need to determine what scenarios are the most executed. If we upload more frequently, EMF Resource/XMI performs better than EMF Resource/Binary. If we do not know that information, EMF Resource/Binary behaves more linearly.

Scalability *Scalability* is defined as the ability of a system to continue to meet its response time as the demand for the software function increases [26].

In this paper, the scalability is studied in terms of the time that a testbed needs to perform the `Upload` scenario as the number of models increases. There, we have studied the scalability by varying the number of models injected into each selected persistence technology. Figures 4 (a) and (b) depict the response time when number of models varies from 1000 to 10,000 models of EPO and LIB, respectively.

As can be observed in Figures 4 (a) and 4 (b), EMF Resource/XMI slightly outperforms EMF Resource/Binary files for EPO models. Teneo/MySQL performs significantly poorly compared to file-based mechanisms for both testbeds. Conversely, for LIB models, the response time taken to upload into EMF Re-



(a) EPO (b) LIB

Fig. 4. Response time for `Upload` scenario by varying the number of models injected into each persistence technology.

source/XMI is less than the taken to upload into EMF Resource/Binary. If we analyse how each persistence technology considered in isolation performs, we observe that the response time is linearly with respect its corresponding model size for all solutions. As in previous experiments, Morsa/MongoDB and Neo4EMF/Neo4j have not been included since prototypes could not be run properly for the very populated testbeds. However, these technologies followed the trend set by Teneo/MySQL in the least populated testbeds (for `Upload` scenario).

This set of experiments shows that the best option for large repositories with small-scale models are those based on file, both XMI and EMF Resource/Binary considering the scalability dimension.

Storage cost We define *storage cost* as the amount of disk space used by the system (or technology) in the persistence. The property can also be used to define the space occupied by the system in memory during runtime. Obviously, this property is related to the capacity cost in the case of persistence technologies deployed on a network (cloud-based deployment), since the transmission time of a persisted repository through a network is directly proportional to its size.

To analyse storage cost, we have carried out two set of experiments. The first one compares the storage cost by injecting the two default testbeds into each persistence technology. As second set, we have also studied the impact in the storage cost of increasing the number of models uploaded.

The results obtained in the first set of experiments are shown in Figure 5. They demonstrate that Morsa/MongoDB consumes a lot more disk space than the rest of technologies, specifically two times larger than Neo4EMF/Neo4j and four times larger than Teneo/MySQL. Nevertheless, Morsa/MongoDB uses the same space disk for both metamodels. We guess that it is due to Morsa internally reserves the disk space by blocks. Although it cannot be seen due to the graphic scale, binary files occupy half disk space used by EMF Resource/XMI files.

Figure 6 shows the second set of experiments. It analyses how each persistence technology grows in size when we injected from 1000 to 10,000 models. As can be observed, the storage cost increases linearly in all cases. As aforementioned, two sets of experiments for Morsa/MongoDB and Neo4EMF/Neo4j were not completed, so they have not been included.

Considering the storage cost, the best persistence mechanism for a very populated repository with small-scale models is EMF Resource/Binary, since much less disk space is used.

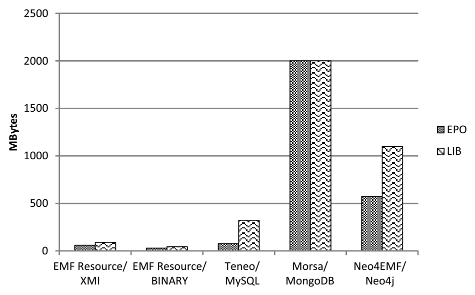
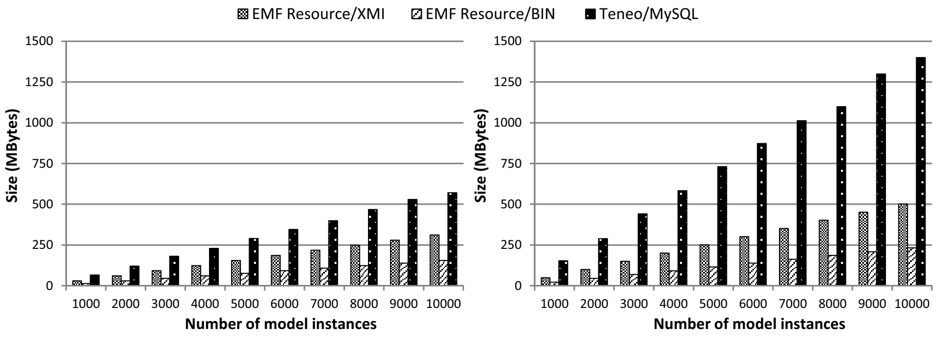


Fig. 5. Storage cost of each persistence technology by injecting default testbeds.



(a) EPO (b) LIB

Fig. 6. Storage Cost by varying the number of models injected into each persistence technology.

4.3 Discussion

The execution of the experiments carried out therefore leads us to determine the best solution for very populated repositories of small/medium-scale models will depend on its use. From the analysis, we have therefore identified the following parameters: i) mean size of metamodels and models, ii) scenarios used frequently; and ii) deployment of the model repository. These three parameters are interrelated, since their variation affects the performance of the others.

Concerning the mean size of metamodels and models, our experiments are focused on small/medium-scale models. As can be observed, file-based solutions outperform database-based ones. Nevertheless, if we compare our results with those obtained in [3, 4, 10] for large-scale models, this kind of file-based mechanisms performs very poorly. Furthermore, since a SAX parser fully reads the XMI file and builds the entire model in memory at once, large models may not be fully kept in memory, causing the parser to overflow the client, as mentioned in [10, 11]. In that cases, a model fragmentation approach as [25] could be considered.

If we take into account what scenario is executed most frequently, response time of `Upload` scenario is slower than `Download`. The former scenario should be scheduled in batch processing at night depending on the persistence solution. For small-scale models, the best solutions are based on XMI and Binary files.

Regarding the deployment of the repository, whether a model repository is deployed onto a network or in the cloud, as a client/server architecture or a web service application, a key aspect is the storage cost of each persistence technology. The time spent to transmit a model to the repository is proportional to its size. As demonstrated, the same models do not consume the same storage cost. In that case, EMF Resource/Binary file as distributed persistence solution is the best option.

We therefore conclude that EMF Resource/Binary file-based mechanism is the best solution for a very populated model repository with small-scale models where the scenario used frequently is `Download` scenario (leaving `Upload` scenario for batch processing).

5 Conclusions and Further Work

Performance constitutes a key aspect of repository-based collaborative modeling to guarantee non-functional requirements, such as response time or scalability. Although there are works in the literature that reporting the comparison of different approaches of model repositories, they only focused on large-scale models. Nevertheless, to the best of our knowledge, we have not identified any work which analyses model repositories to persist large numbers of small-scale models.

In this paper, we explore different persistence technologies and compare from performance perspective. For this purpose, we have designed and implemented a model generic repository to abstract different persistence mechanisms and different modeling languages. This generic architecture allows each persistence technologies to be compared under the same situations.

For future work we have identified a further promising line of research. We aim to implement not only other persistence technologies in our generic architecture, such as XML-native database or EMFStore [14]; but also, other performance issues, such as concurrency or Brewer's Theorem (consistency, availability, partition tolerance). In addition, we have planned to include new functionalities, such as a search engine, a version control system, merge functions and resolution of cross-document dependencies between persisted models. Finally, the obtained results in this paper allow persistence technology to be selected to achieve a collaborative MDD framework for small/medium-scale models.

Acknowledgments The work for this paper was supported by funding from ISBAN and PRODUBAN, under the Center for Open Middleware initiative.

References

1. *Apache Subversion*, 2015. Available at: `https://subversion.apache.org/`.
2. K. Banker. *MongoDB in Action.* Manning Publications Co., 2011.
3. K. Barmpis and D. S. Kolovos. Comparative Analysis of Data Persistence Technologies for Large-scale Models. In *Procs. of the 2012 Extreme Modeling Workshop*, XM '12, pages 33–38, 2012.
4. A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay. Neo4EMF, A Scalable Persistence Layer for EMF Models. In *Modelling Foundations and Applications*, volume 8569 of *LNCS*, pages 230–241. Springer-Verlag, 2014.
5. X. Blanc, M.-P. Gervais, and P. Sriplakich. Model Bus: Towards the Interoperability of Modelling Tools. In *Model Driven Architecture*, volume 3599 of *LNCS*, pages 17–32. Springer Berlin Heidelberg, 2005.
6. H. Brunelière, J. Cabot, S. Drapeau, F. Somda, W. Piers, J. D. Villa Calle, and J.-C. Lafaurie. MDE Support for Enterprise Architecture in an Industrial Context: the TEAP Framework Experience. In *TowArds the Model DrIveN Organization (AMINO 2013) workshop - a MODELS 2013 Satellite Event*, 2013.
7. M. A. Almeida da Silva, A. Abherve, and A. Sadovykh. From the Desktop to the Multi-clouds: The Case of ModelioSaaS. In *15th Int. Symp. on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2013*, pages 462–469. IEEE Computer Society, 2013.

8. M. Dirix, A. Muller, and V. Aranega. Genmymodel: An online uml case tool. *Joint Proceedings of Tools, Demos & Posters*, page 14, 2013.
9. A. El Kouhen, C. Dumoulin, S. Gerard, and P. Boulet. Evaluation of Modeling Tools Adaptation. Technical report, 2012.
10. J. Espinazo Pagán, J. Sánchez Cuadrado, and J. García Molina. Morsa: A Scalable Approach for Persisting and Accessing Large Models. In *Model Driven Engineering Languages and Systems*, volume 6981 of *LNCS*, pages 77–92. Springer-Verlag, 2011.
11. J. Espinazo Pagán, J. Sánchez Cuadrado, and J. García Molina. A repository for scalable model management. *Software & Systems Modeling*, pages 1–21, 2013.
12. Fraunhofer. *ModelBus*, 2015. Available at: `http://www.modelbus.org/`.
13. R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley professional computing. John Wiley, 1991.
14. M. Koegel and J. Helming. EMFStore: A Model Repository for EMF Models. In *Procs. of the 32Nd ACM/IEEE Int. Conf. on Software Engineering - Volume 2*, ICSE '10, pages 307–308. ACM, 2010.
15. MetaCase. *MetaEdit+ Domain-Specific Modeling Environment* . Available at: `https://http://www.metacase.com/products.html`.
16. MongoDB Developers. *MongoDB*. Available at: `http://www.mongodb.org/`.
17. Neo4j Developers. *Neo4j*. Available at: `http://neo4j.com/`.
18. No Magic. *MagicDraw*, 2015. Available at: `http://www.nomagic.com/products/magicdraw.html`.
19. OMG. *A UML profile for Modeling and Analysis of Real Time Embedded Systems (MARTE)*, 2011. Version 1.1 Available at: `http://www.omg.org/spec/MARTE/1.1/`.
20. OMG. *Unified Modeling Language (UML)*, 2012. Version 2.4.1 Available at: `http://www.omg.org/spec/UML/2.4.1/`.
21. OMG. *Business Process Model And Notation (BPMN)*, 2013. Version 2.0.2 Available at: `http://www.omg.org/spec/BPMN/2.0.2/`.
22. OMG. *MetaObject Facility (MOF)*, 2014. Version 2.4.2 Available at: `http://www.omg.org/spec/MOF/2.4.2`.
23. OMG. *XML Metadata Interchange (XMI)*, 2014. Version 2.4.2 Available at: `http://www.omg.org/spec/XMI/2.4.2/`.
24. Oracle. *MySQL*. Available at: `http://www.mysql.com/`.
25. M. Scheidgen, M. Zubow, J. Fischer, and T. H. Kolbe. Automated and transparent model fragmentation for persisting large models. In *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, pages 102–118, 2012.
26. C. U. Smith and L. G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison–Wesley, 2002.
27. SOFTEAM. *Modelio*, 2015. Available at: `http://www.modelio.org`.
28. D. Steinberg, F. Budinsky, M. Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.
29. The Eclipse Foundation. *Teneo*. Available at: `http://eclipse.org/modeling/emft/?project=teneo`.
30. The Eclipse Foundation. *The CDO Model Repository*. Available at: `https://eclipse.org/cdo/`.
31. The Eclipse Foundation. *MoDisco Eclipse Project*, 2014. Available at: `https://eclipse.org/MoDisco/`.