Stress-Testing Centralised Model Stores

Antonio Garcia-Dominguez, Konstantinos Barmpis,
Dimitrios S. Kolovos, Ran Wei, and Richard F. Paige

Department of Computer Science,
University of York, United Kingdom
{firstname.lastname}@york.ac.uk

Abstract. One of the current challenges in model-driven engineering is enabling effective collaborative modelling. Two common approaches are either storing the models in a central repository, or keeping them under a traditional file-based version control system and build a centralized index for model-wide queries. Either way, special attention must be paid to the nature of these repositories and indexes as networked services: they should remain responsive even with an increasing number of concurrent clients. This paper presents an empirical study on the impact of certain key decisions on the scalability of concurrent model queries, using an Eclipse Connected Data Objects model repository and a Hawk model index. The study evaluates the impact of the network protocol, the API design and the internal caching mechanisms and analyzes the reasons for their varying performance.

1 Introduction

Model-driven engineering (MDE) has received considerable attention due to its demonstrated benefits of improving productivity, quality and maintainability. However, industrial adoption has ran into various challenges regarding the current maturity and scalability of MDE.

In their 2012 study [1], Mohagheghi et. al. interviewed four companies and noted that they considered that the tools at the time did not scale to the large projects that would merit the use of MDE. Several ways in which MDE practice could learn from widely-used programming environments to handle large models better were pointed out in [2], with a strong focus on the need for modularity in modelling languages to improve scalability and simplify collaboration. Three categories of scalability issues in MDE were identified in [3]: model persistence issues, model querying and transformation issues, and collaborative modelling issues.

Focusing on collaborative modelling, one common approach is storing the models in a *model repository* that keeps track of revisions and resolves conflicts between users. Another approach is using a file-based version control system for storage and version control and creating a *model index* that can efficiently answer model-wide queries and provide integrated views of all the model fragments. These tools are typically deployed as a service within a network, in order to make the models available from any device within the organisation.

There are strong implementations of both approaches, but little attention has been paid to their inherent nature as networked services: most studies have either focused on local scenarios where the client and the server reside in the same machine, or considered only the simple case with a single remote user. It can be argued that even with the right features, a collaborative system that is not responsive under load would not be widely adopted.

In this empirical study we evaluate the impact of several design decisions in one functionality common to both model repositories (e.g. CDO) and model indices (e.g. Hawk): querying remote models. The study aims to inform developers and end users of remote model stores on the real tradeoffs between several common choices: which network protocol to use, how to design the API and which types of caches are most effective.

The rest of this work is structured as follows: Section 2 provides a discussion on existing work on model repositories and model indices, Section 3 introduces the research questions and the design of the experiment, Section 4 discusses the obtained results and Section 5 presents the conclusions and future lines of work.

2 Background and Related Work

Persisting and managing large models has been extensively investigated over the past decade. This section presents the main state-of-the-art tools and technologies and briefly describes the two tools used in this empirical study.

2.1 File-Based Model Persistence

One of the most common approaches to storing models is serializing them to a textual file-based form. Tools like the Eclipse Modeling Framework (EMF) [4], ModelCVS [5], Modelio[1] and MagicDraw[2] all use XML-based model serialization. While this approach offers a structured platform-independent way for storing models, it has been shown by various works such as [3,6] to lack scalability as it requires loading the entire text file in order to retrieve any information needed from the model.

2.2 Database-Backed Model Persistence

In light of the scalability limitations resulting from storing models as text files, various database-backed model persistence formats have been proposed. Teneo/ Hibernate[3] allows EMF models to be stored in relational databases. NeoEMF [6] and MongoEMF[4] all use NoSQL databases to store EMF models. Database persistence allows for partial loading of models as only accessed elements have

[1] `https://www.modelio.org/`
[2] `http://www.nomagic.com/products/magicdraw.html#Collaboration`
[3] `http://wiki.eclipse.org/Teneo/Hibernate`
[4] `https://github.com/BryanHunt/mongo-emf/wiki`

to be loaded in each case. Furthermore, such technologies can leverage the use of database indices and caches for improving element lookup performance as well as query execution time.

2.3 Model Repositories

When collaborative modeling is involved, simply storing models in a scalable form such as inside a database stops being sufficient; in this case issues such as collaborative access and versioning need to also be considered. Examples of model repository tools are Morsa [7], ModelCVS[5], Connected Data Objects (CDO)[6], EMFStore [8], Modelio, MagicDraw and MetaEdit+[7]. Model repositories offer capabilities for allowing multiple developers to manage models stored in a centralized repository by ensuring the models remain in a consistent state, while persisting the models in a scalable form, such as in a database.

CDO in particular is one of the most mature solutions, having been developed for over 7 years as an Eclipse project and being currently maintained by Obeo [8]. It implements a pluggable storage architecture, being able to use various solutions such as relational databases (H2, MySQL) or document-oriented databases (MongoDB), among others. CDO includes Net4j, a messaging library that provides bidirectional communications over TCP, HTTP and in-memory connections, and uses it to provide a remote API that exposes remote models as EMF resources. In addition to storing models, CDO includes a CDOQuery API that makes it possible to run queries remotely on the server and retrieve directly the results, reducing the necessary bandwidth.

2.4 Heterogeneous Model Indexing

An alternative to using model repositories for storing models used in a collaborative environment is to store them as file-based models in a classical version control system. As discussed in [9] this leverages the benefits of widely-used file-based version control systems such as SVN and Git, but retains the issues file-based models face. To address this issue a model indexer can be introduced that monitors the models and indices them in a scalable model index. The model index is synchronized with the latest version of the models in the repository and can be used to perform efficient queries on them, without having to check them out or load them into memory.

Hawk is an example of such a technology: it can maintain a graph database with the contents of one or more version control systems and perform very efficient queries on them. More recently, Hawk has been embedded into a server that provides a service-oriented API based on the Apache Thrift[9] library.

[5] `http://www.modelcvs.org/versioning/index.html`
[6] `http://wiki.eclipse.org/CDO`
[7] `http://www.metacase.com/`
[8] As stated in `http://projects.eclipse.org/projects/modeling.emf.cdo`
[9] `http://thrift.apache.org/`

3 Experiment Design

As mentioned in the introduction, effective collaborative modelling requires not only having the right features, but also making sure that the system stays responsive as the number of clients increases. This section presents the design of an empirical study that evaluates the impact of several factors in the performance of remote queries on a model repository (CDO) and a model index (Hawk).

3.1 Research Questions

RQ1. *What is the impact of the network protocol on remote query times and throughputs?*

In order to connect to a remote server, two of the most popular options are using raw TCP connections (for the sake of performance and flexibility) or sending HTTP messages (for compatibility with web browsers and interoperability with proxies and firewalls). Both Hawk and CDO support TCP and HTTP.

Properly configured HTTP servers and clients can reuse the underlying TCP connections with HTTP 1.1 pipelining and avoid repeated handshakes, but the additional overhead imposed by the HTTP fields may still impact the raw performance of the tool.

RQ2. *What is the impact of the design of the remote query API on remote query times and throughputs?*

Application protocols for network-based services can be stateful or stateless. Stateful protocols require that the server keeps track of part of the state of the client, while stateless protocols do not have this requirement. In addition, the protocol may be used mostly for transporting opaque blocks of bytes between server and client, or it might have a well-defined set of operations and messages.

While a stateful protocol may be able to take advantage of the shared state between the client and server, a stateless protocol is generally simpler to implement and use. Service-oriented protocols need to also take into account the granularity of each operation: "fine" operations that do only one thing may be easier to recombine, but they will require more invocations than "coarse" operations that perform a task from start to finish.

CDO implements a stateful protocol on top of the Net4j library, which essentially consists of sending and receiving buffers of bytes across the network. On the other hand, Hawk implements a stateless service-oriented API on top of the Apache Thrift library, exposing a set of specific operations (e.g. "query", "send object" or "register metamodel"). The Hawk API is generally coarse: most queries only require one pair of HTTP request/response messages.

While the stateful CDO clients and servers may cooperate better with each other, the simpler and less granular API in Hawk may reduce the total network roundtrip for a query by exchanging less messages.

RQ3. *What is the impact of the internal caching and indexing mechanisms on remote query times and throughputs?*

Database-backed systems generally implement various caching strategies to keep the most frequently accessed data in memory, away from slow disk I/O. At the very least, the DBMS itself will generally keep its own cache, but the system might use additional memory to cache especially important subsets or to keep them in a form closer to how it is consumed.

Another common strategy is to prepare indices in advance, speeding up queries. DBMSs already provide indices for common concepts such as primary keys and unique values, but these systems may add their own application-specific indices that precompute parts of the queries to be run.

3.2 Experiment Setup

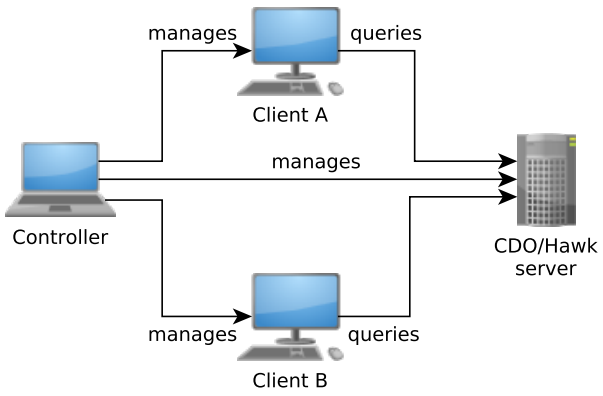


Fig. 1. Network diagram for the experimental setup

In order to provide answers for the above research questions, a networked environment was set up to emulate increasing numbers of clients interacting with a model repository (CDO 4.4.1.v20150914-0747) or a model index (Hawk 1.0.0. 201602231713) and collect query response times. The environment is outlined in Figure 1, and consists of the following:

– **One "Controller" machine** that supervises the other machines through SSH connections managed with the Fabric Python library[10]. It is responsible for starting and stopping the client and server processes, monitoring their execution, and collecting the measured values. It does not run any queries itself, so it has no impact on the obtained results.

[10] http://www.fabfile.org/

- **Two "Client" machines** that invoke the queries on the server, fetch the results and measure query response times. The two client machines were running Ubuntu Linux 14.04.3, Linux 3.19.0-47-generic and Oracle Java 8u60 on an Intel Core i5 650 CPU, 8GiB of RAM and a 500GB SATA3 hard disk. The client machines had two client programs installed: one for CDO and one for Hawk. Only one of these programs ran at a time. Each of these programs received the address of the server to connect to, the size of the Java fixed thread pool to be used, the number of queries to be distributed across these threads and a template for the query to be run.
- **One "Server" machine** that hosts the CDO model repository and the Hawk model index, and provides TCP and HTTP ports exposing the standard CDO and Hawk APIs for remote querying. The server machine had the same configuration as the client machines.
  The server machine had two server programs installed: one for CDO and one for Hawk. Again, only one of these programs ran at a time. Both server programs were Eclipse products based on Eclipse Mars and used the same embedded HTTP server (Eclipse Jetty 9.2.13). Both systems were configured to use up to 4096MB of memory (`-Xmx4096m -Xms2048m`)[11].
  In particular, the CDO server was based on the standard CDO server product, with the addition of the experimental HTTP Net4j connector. No other changes were made to the CDO configuration. The database backend was H2, the most mature and feature-complete option at the time of writing.
- **One 100Mbps network switch** that connected all machines together in an isolated local area network.

As the study was intended to measure query performance results with increasing numbers of concurrent users, the client programs were designed to first warm up the servers into a *steady state*. Query time was measured as the time required to connect to the server, run the query on the server and retrieve the model element identifiers of the results over the network. Queries would be run 1000 times in all configurations, to reduce the impact of variations due to external factors (CPU and I/O scheduling, Java just-in-time recompilation, disk caches, virtual memory and so on).

Several workloads were defined. The lightest workload used only 1 client machine with 1 thread running 1000 queries in sequence. The other workloads used 2 client machines, each running 500 queries, with increasingly large pools of 2, 4, 8, 16, and 32 threads. These workloads would simulate between 1 and 64 clients running queries concurrently.

3.3 Queries Under Study

After defining the research questions and preparing the environment, the next step was to populate CDO and Hawk with the contents to be queried, and to

[11] The Neo4j performance guide suggests this amount for a system with up to 100M nodes and 8GiB RAM, to allow the OS to keep the graph database in its disk cache.

Listing 1. GraBaTs query written in OCL (OQ) for evaluating CDO.

```
1  DOM::TypeDeclaration.allInstances()→select(td |
2      td.bodyDeclarations→selectByKind(DOM::MethodDeclaration)
3        →exists(md : DOM::MethodDeclaration |
4          md.modifiers
5            →selectByKind(DOM::Modifier)
6            →exists(mod : DOM::Modifier | mod.public)
7          and md.modifiers
8            →selectByKind(DOM::Modifier)
9            →exists(mod : DOM::Modifier | mod._static)
10         and md.returnType.oclIsTypeOf(DOM::SimpleType)
11         and md.returnType.oclAsType(DOM::SimpleType).name.fullyQualifiedName
12           = td.name.fullyQualifiedName))
```

write equivalent queries in their back-end independent languages: OCL for CDO and the Epsilon Object Language [10] (EOL) for Hawk.

CDO and Hawk were populated through the MoDisco use case titled *SharenGo Java Legacy Reverse-Engineering*[12] that was presented at the GraBaTs 2009 contest [11]. This use case involved reverse-engineering increasingly large Java codebases in order to extract knowledge models. The largest codebase in the case study was selected, covering all the `org.eclipse.jdt` projects and producing over 4.9 million model elements. The H2 and Neo4j databases in CDO and Hawk grew to 1.4GB and 1.9GB, respectively.

These model elements conformed to the Java Development Tools AST (JDTAST) metamodel, which is described in works such as [3] or [7]. Some of the types within the JDTAST metamodel include the TypeDeclarations that represent Java classes and interfaces, the MethodDeclarations that represent Java methods, and the Modifiers that represent Java modifiers on the methods (such as `static` or `public`).

Based on these types, task 1 in the GraBaTs 2009 contest required defining a query (from now on referred to as the GraBaTs query) that would locate all possible applications of the Singleton design pattern in Java [12]. In other words, it would have to find all the TypeDeclarations that had at least one MethodDeclaration with `public` and `static` modifiers that returned an instance of the same TypeDeclaration.

To evaluate CDO, the GraBaTs query was written in OCL as shown in Listing 1. The query (named OQ after "OCL query") filters the TypeDeclarations by iterating through their MethodDeclarations and their respective Modifiers.

To evaluate Hawk, we used the three EOL implementations of the GraBaTs query of our previous work [13]. The first version of the query ("Hawk query 1"

[12] http://www.eclipse.org/gmt/modisco/useCases/JavaLegacyRE/

Listing 2. GraBaTs query written in EOL (HQ1) for evaluating Hawk.

```
1 return TypeDeclaration.all.select(td|
2   td.bodyDeclarations.exists(md:MethodDeclaration|
3     md.returnType.isTypeOf(SimpleType)
4     and md.returnType.name.fullyQualifiedName == td.name.fullyQualifiedName
5     and md.modifiers.exists(mod:Modifier|mod.public==true)
6     and md.modifiers.exists(mod:Modifier|mod.static==true)));
```

Listing 3. GraBaTs query written in EOL (HQ2) using derived attributes on the MethodDeclarations for evaluating Hawk.

```
1 return MethodDeclaration.all.select(md |
2   md.isPublic and md.isStatic and md.isSameReturnType
3 ).collect( td | td.eContainer ).asSet;
```

or HQ1, shown in Listing 2) is a translation of OQ to EOL, and follows the same approach.

The second version (HQ2), shown in Listing 3, assumed that the user instructed Hawk to extend MethodDeclarations with three derived attributes: *isStatic* (the method has a `static` modifier), *isPublic* (the method has a `public` modifier), and *isSameReturnType* (the method returns an instance of its TypeDeclaration). The query starts off from the MethodDeclarations so Hawk can take advantage of the fact that derived attributes are also indexed, so Hawk can use lookups instead of iterations to find the methods of interest. A detailed discussion about how derived attributes are declared in Hawk and how they are incrementally re-computed upon model changes is available in our previous works [9,14].

The third version (HQ3), shown in Listing 4, assumed instead that Hawk extended TypeDeclarations with the *isSingleton* derived attribute, setting it to true when the TypeDeclaration has a `static` and `public` MethodDeclaration returning an instance of itself. This derived attribute eliminates one more level of iteration, so the query only goes through the TypeDeclarations.

The GraBaTs query has been translated to 1 OCL query (OQ) and 3 possible EOL queries (HQ1 to HQ3). It must be noted that since CDO does not support derived attributes like Hawk, it was not possible to rewrite OQ in the same way as HQ1. Since the same query would be repeatedly run in the experiments, the

Listing 4. GraBaTs query written in EOL (HQ3) using derived attributes on the TypeDeclarations for evaluating Hawk.

```
1 return TypeDeclaration.all.select(td|td.isSingleton);
```

authors inspected the code of CDO and Hawk to ensure that neither tool cached the results of the queries themselves: this was verified by re-running the queries while adding unique trivially true conditions, and comparing execution times.

4 Results and Discussion

The previous section described the research questions to be answered, the environment that was set up for the experiment and the queries to be run. This section will present the obtained results, answer the research questions (with the help of additional data in some cases) and discuss potential threats to the validity of the work.

4.1 Measurements Obtained

The obtained results for OQ, HQ1, HQ2 and HQ3 are shown in Figures 2 to 5. These are notched box plots that show the distribution of query execution times over $n = 1000$ samples, using a logarithmic scale on the y axes. Results are faceted over the total number of client threads (from 1 to 64) and then separated by protocol (TCP or HTTP). Each figure also includes a data table with the median query execution times in milliseconds by thread count and protocol, for direct numerical comparison between the alternatives. Shapiro-Wilk tests rejected the null hypothesis ("the sample comes from a normal distribution") with p-values < 0.01 for almost all combinations of query, protocol and thread count. This was confirmed with Q-Q plots as well. For the purposes of this study, we will assume that the query execution times are not normally distributed and use non-parametric tests.

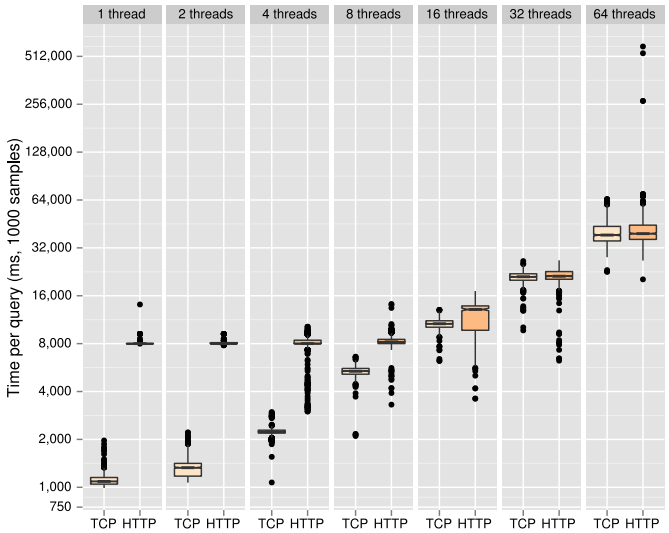
The box plots include dots with the outliers detected among the 1000 samples: these are the values above and below 1.5 times the interquartile distance (IQR) from the third and first quartile, respectively. These outliers are assumed to originate from thread ramp-up and ramp-down and from variations in the I/O and CPU schedulers of the operating system and the underlying caches.

While they are not formal statistical tests, the notches serve as approximate confidence intervals for the real median, based on $median \pm IQR/\sqrt{n}$ [15].

Most configurations (CDO with TCP, all versions of Hawk) ran all queries correctly, producing the expected 164 results. However, CDO with HTTP resulted in several queries failing to respond or returning incorrect results with 4 threads (3 out of 1000), 8 (8), 16 (22), 32 (18) and 64 (2). CDO with HTTP also produced four especially notable outliers: 2 took 268.5s, one took 535.5s and the last one took 590.5s.

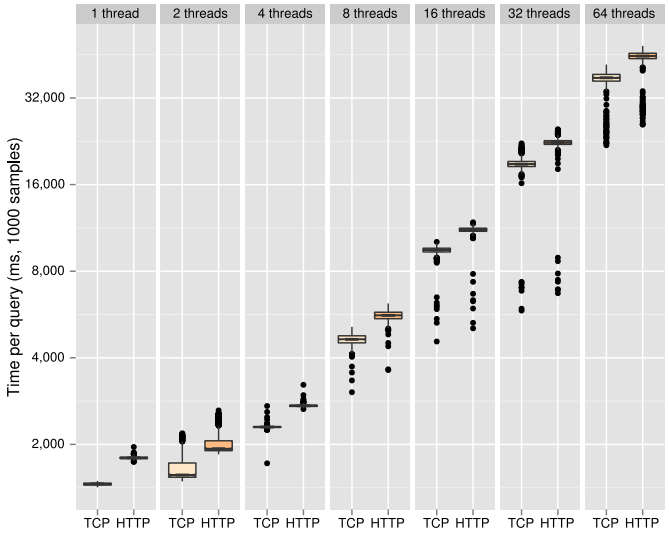
4.2 RQ1: Impact of Protocol

Regarding the impact of the protocol on the performance and throughput of the remote queries, both CDO and Hawk confirm that there is a certain overhead involved in using HTTP rather than TCP. However, the actual overhead is very different depending on the tool and the query:



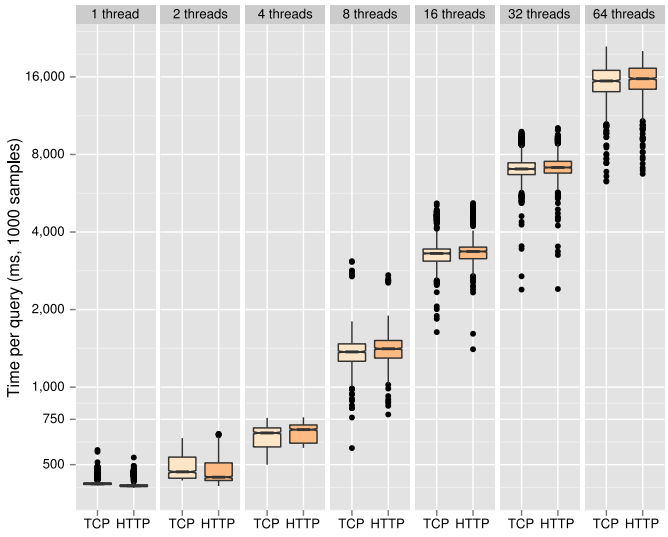
	1	2	4	8	16	32	64
Median TCP (ms)	1,088	1,328	2,233	5,365	10,672	21,068	38,525
Median HTTP (ms)	8,004	8,004	8,010	8,207	13,115	21,229	39,328

Fig. 2. OQ execution times (CDO, OCL)



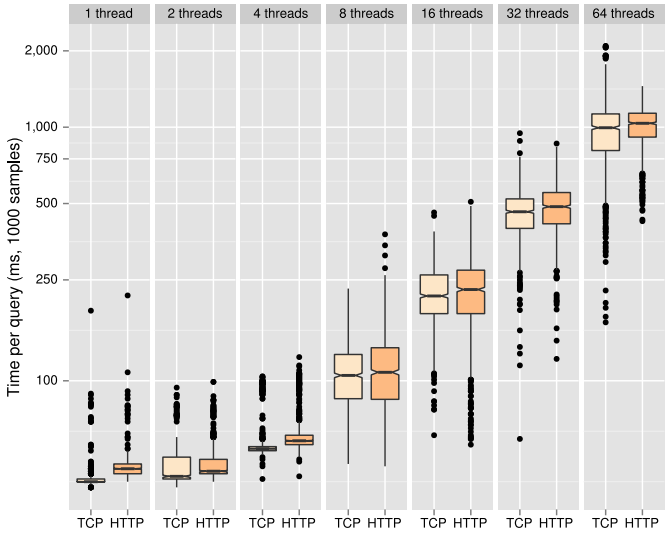
	1	2	4	8	16	32	64
Median TCP (ms)	1,459	1,565	2,299	4,636	9,478	18,834	37,629
Median HTTP (ms)	1,797	1,932	2,724	5,616	11,159	22,407	44,761

Fig. 3. HQ1 execution times (Hawk, EOL, no derived attributes)



	1	2	4	8	16	32	64
Median TCP (ms)	421	469	664	1,370	3,303	7,028	15,432
Median HTTP (ms)	414	447	684	1,409	3,361	7,121	15,748

Fig. 4. HQ2 execution times (Hawk, EOL, extended MethodDeclarations)



	1	2	4	8	16	32	64
Median TCP (ms)	40	42	54	105	216	464	995
Median HTTP (ms)	45	44	58	108	229	486	1,036

Fig. 5. HQ3 execution times (Hawk, EOL, extended TypeDeclarations)

- OQ (Figure 2) shows that CDO over HTTP has a striking overhead with low thread counts, taking nearly 8 times as much time per query as CDO over TCP for only 1 client thread. Kruskall-Wallis tests confirm significant differences for all thread counts (all p-values are < 0.01).
- HQ1 (Figure 3) over HTTP has a consistent overhead over using TCP, but not as striking as CDO's case for OQ. Comparing the medians between TCP and HTTP, using HTTP increased query times between 17.74% and 23.45%. Kruskall-Wallis tests confirmed significant differences for all numbers of threads, with p-values < 0.01.
- The lower query times for HQ2 (Figure 4) and HQ3 (Figure 5) reduce CPU contention and make HTTP and TCP more alike. However, there are still significant differences, with Kruskall-Wallis p-values slightly higher but consistently below 0.02 for all numbers of threads.

4.3 RQ2: Impact of API Design

One striking observation from RQ1 was that CDO over HTTP had much higher overhead than Hawk over HTTP. Comparing the medians of OQ and HQ1 with 1 client thread, CDO+HTTP took 635.66% longer than CDO+TCP, while Hawk+HTTP only took 23.17% longer than Hawk+TCP. This contrast showed that CDO and Hawk used HTTP to implement their APIs very differently.

To clarify this issue, the Wireshark packet sniffer was used to capture the communications between the server and the client for one invocation of OQ and HQ1. These captures showed quite different approaches for an HTTP-based API:

- **CDO** involved exchanging 58 packets (10203 bytes), performing 11 different HTTP requests. Many of these requests were very small and consisted of exchanges of byte buffers between the server and the client, opaque to the HTTP servlet itself.
  Most of these requests were either within the first second of the query execution time or within the last second. There was a gap of approximately 6 seconds between the first group of requests and the last group. Interestingly, the last request before the gap contained the OCL query and the response was an acknowledgment from CDO. On the first request after the gap, the client sent its session ID and received back the results from the query.
  The capture indicates that these CDO queries are asynchronous in nature: the client sends the query and eventually gets back the results. While the default Net4j TCP connector allows the CDO server to talk back to the client directly through the connection, the experimental HTTP connector relies on polling for this task. This has introduced unwanted delays in the execution of the queries. The result suggests that an alternative solution for this bidirectional communication would be advisable, such as WebSockets.
- **Hawk** involved exchanging 14 packets (2804 bytes), performing 1 HTTP request and receiving the results of the query in the same response. Since its API is stateless, there was no need to establish a session or keep a bidirectional server–client channel: the results were available as soon as possible.

While this synchronous and stateless approach is much simpler to implement and use, it does have the disadvantage of making the client block until all the results have been produced. Future versions of Hawk could also implement asynchronous querying in a similar way to what was suggested for CDO.
One side note is that Hawk required using much less bandwidth than CDO: this was due to a combination of using fewer requests, using `gzip` compression on the responses and taking advantage of the most efficient binary encoding available in Apache Thrift (*Tuple*).

In summary, CDO and Hawk use HTTP in very different ways. The CDO API is stateful and consists of exchanging pending buffers between server and client periodically: queries are asynchronous and results are sent back through polling. The Hawk API is service-oriented, stateless and synchronous: query results are sent back immediately. These results suggest that systems may benefit from supporting both synchronous querying (for small or time-sensitive queries) and asynchronous querying (for large or long-running queries), and that asynchronous querying must be carefully implemented to avoid unnecessary delays.

4.4 RQ3: Impact of Tool Internals

This section will focus on the results from the TCP variants, since they outperformed all the HTTP variants in the previous tests. Comparing the results produced by the four queries, there are several key observations to make:

O1. *OQ ran faster than HQ1 with 1 and 2 threads.*
O2. *HQ1 ran faster than OQ between 8 and 64 threads.*

O1 was somewhat unexpected: it was assumed that the join-free adjacency of the Neo4j graph database used in Hawk would give it an edge over the default H2 relational backend in CDO. Enabling the SQL trace log in CDO showed that after the first execution of OQ, later executions only performed one SQL query to verify if there were any new instances of TypeDeclaration.

Previous tests had already rejected the possibility that CDO was caching the query results. Instead, an inspection of the CDO code revealed a collection of generic caches. Among others, CDO keeps a CDOExtentMap from EClasses to all their EObject instances, and also keeps a CDORevisionCache with the various versions of each EObject. In comparison, Hawk only uses the DBMS cache and an internal type cache, so it needs to retrieve the objects from the DBMS every time they are needed.

On the other hand, O2 shows that the lighter caching and Neo4j backend of Hawk allow it to scale better with demand: with 8 threads, the median time with Hawk is 4.64s instead of the 5.37s of CDO.

O3. *HQ2 ran faster than HQ1 and OQ for all thread counts.*
O4. *HQ3 ran faster than HQ2 for all thread counts.*

O3 and O4 confirm the findings of our previous work in scalable querying [9,14]: adding derived attributes to reduce the levels of iteration required in a query speeds up running times by orders of magnitude, while adding minimal overhead due to the use of incremental updating. These derived attributes can be seen as application-specific caches that precompute parts of a query, unlike the application-agnostic caches present in CDO.

In particular, HQ3 takes two orders of magnitude less time than OQ and HQ1 for 1 thread, and is still faster than the best results of OQ and HQ1 even when handling 64 threads. While the *isSingleton* derived attribute in HQ3 might be considered too specialized for most cases, the *isStatic*, *isPublic* and *isSameReturnType* attributes in HQ2 are more generally useful and still produce significant time savings over OQ and HQ1.

4.5 Limitations and Threats to Validity

The presented study has several limitations that may threaten the internal and external validity of the results. Regarding the internal validity of the results:

- There is a possibility that CDO or Hawk could have been configured or used in a more optimal way. Since the authors developed Hawk, this may have allowed them to fine-tune Hawk better than CDO. However, the servers did not show any undesirable virtual memory usage, excessive garbage collection or disk I/O. The H2 backend was chosen for CDO due to its maturity in comparison to the other backends, and the Neo4j backend has consistently produced the best results for Hawk according to previous work. Finally, the authors contacted the CDO developers regarding how to compress responses and limit results by resource, to make it more comparable with Hawk, and were informed that these were not supported yet[13].
- The queries for CDO and Hawk were written in different languages, so part of the differences in their performance may be due to the languages and not the systems themselves. The aim in this study was to use the most optimized language for each system, since Hawk does not support OCL and CDO does not support EOL. Analytically, we do not anticipate that this is likely to have a strong impact on the obtained results as both languages are very similar in nature and are executed via mature Java-based interpreters. A future study could extend CDO or Hawk to fully address this issue.

As for the external validity of the results, this first study has not considered running several different queries concurrently, and has only considered one particular configuration for Hawk and CDO. While this configuration would be quite typical in most organisations, further studies are needed that mix different queries running in different models concurrently, and configure Hawk and CDO with different backends, memory limits, and model sizes.

[13] https://www.eclipse.org/forums/index.php?t=rview&goto=1722258
https://www.eclipse.org/forums/index.php?t=rview&goto=1722096

5 Conclusions and Further Work

The results from the study suggest that the network protocol can have very different impacts on the performance of the model repository or model index, depending on how it is used: while CDO over HTTP had a dramatic overhead of 640%, Hawk over HTTP had a more reasonable 20%. The 20% overhead is a reasonable price to pay for the simpler operation across firewalls and its availability from Web browsers. A packet capture revealed that the problem with CDO over HTTP was the naïve way in which server-to-client communications had been implemented, which used simple polling instead of state-of-the-art approaches such as WebSockets. The study also showed that while CDO's extensive application-agnostic caching was somewhat faster than Hawk with no derived attributes, Hawk with derived attributes (a form of application-specific caching) could outperform CDO by two orders of magnitude and still be faster for large numbers of concurrent clients. As a general conclusion, this study confirms that the apparent performance of a highly efficient model access API can be severely degraded by minor details in the communications layer, and that having a few and well-placed application-specific caches can be much more effective than a comprehensive set of application-agnostic caches.

We plan to extend the present study to cover more situations, with more configurations of Hawk and CDO, a wider assortment of queries and a larger number of clients. Another direction of future work is analyzing the queries to split the work in a query efficiently between the client and the server, using the server for model retrieval and using the client to transform the retrieved values.

Acknowledgments

This research was part supported by the EPSRC, through the Large-Scale Complex IT Systems project (EP/F001096/1) and by the EU, through the MONDO FP7 STREP project (#611125).

References

1. Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel A. Fernandez. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical Software Engineering*, 18(1):89–116, January 2012.
2. Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Scalability: The Holy Grail of Model Driven Engineering. In *Proc. Workshop on Challenges in MDE, collocated with MoDELS '08, Toulouse, France*, 2008.
3. Konstantinos Barmpis and Dimitrios S. Kolovos. Evaluation of contemporary graph databases for efficient persistence of large-scale models. *Journal of Object Technology*, 13-3:3:1–26, July 2014. DOI 10.5381/jot.2014.13.3.a3.
4. Marcelo Paternostro Dave Steinberg Frank Budinsky and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2008.

5. G. Kramler, G. Kappel, T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. Towards a semantic infrastructure supporting model-based tool integration. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, GaMMa '06, pages 43–46, New York, NY, USA, 2006. ACM.
6. Abel Gómez, Massimo Tisi, Gerson Sunyé, and Jordi Cabot. Map-based transparent persistence for very large models. In Alexander Egyed and Ina Schaefer, editors, *Fundamental Approaches to Software Engineering*, volume 9033 of *Lecture Notes in Computer Science*, pages 19–34. Springer Berlin Heidelberg, 2015.
7. Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. A repository for scalable model management. *Software & Systems Modeling*, pages 1–21, 2013. DOI 10.1007/s10270-013-0326-8.
8. Maximilian Koegel and Jonas Helming. EMFStore: a model repository for EMF models. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 307–308. ACM, 2010.
9. Konstantinos Barmpis, Seyyed Shah, and Dimitrios S. Kolovos. Towards incremental updates in large-scale model indexes. In *Proceedings of the 11th European Conference on Modelling Foundations and Applications. ECMFA'15*, July 2015.
10. Kolovos, D.S., Paige, R.F. and Polack, F.A. The Epsilon Object Language. In *Proc. European Conference in Model Driven Architecture (EC-MDA) 2006*, volume 4066 of *LNCS*, pages 128–142, Bilbao, Spain, July 2006.
11. GraBaTs. 5th Int. Workshop on Graph-Based Tools, 2009. `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/` (last checked: February 29th, 2016).
12. Jean-Sebastien Sottet and Frédéric Jouault. Program comprehension. In *Proc. 5th Int. Workshop on Graph-Based Tools*, 2009. `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2009reverseengineering.pdf` (last checked: February 29th, 2016).
13. Konstantinos Barmpis and Dimitrios S. Kolovos. Towards Scalable Querying of Large-Scale Models. In *Modelling Foundations and Applications - 10th European Conference, ECMFA 2014, Held as Part of STAF 2014, York, UK, July 21-25, 2014. Proceedings*, pages 35–50, 2014.
14. Konstantinos Barmpis and Dimitrios S. Kolovos. Towards scalable querying of large-scale models. In *Proceedings of the 10th European Conference on Modelling Foundations and Applications. ECMFA'14*, July 2014.
15. John M. Chambers, William S. Cleveland, Paul A. Tukey, and Beat Kleiner. *Graphical Methods for Data Analysis*. Duxbury Press, Boston, USA, first edition, 1983. ISBN 978-0-534-98052-8.