Stream my Models: Reactive Peer-to-Peer Distributed Models@run.time

Thomas Hartmann, Assaad Moawad, Francois Fouquet,
Gregory Nain, Jacques Klein, and Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust (SnT),
University of Luxembourg
Luxembourg
Email: firstName.lastName@uni.lu

***Abstract*—The models@run.time paradigm promotes the use of models during the execution of cyber-physical systems to represent their context and to reason about their runtime behaviour. However, current modeling techniques do not allow to cope at the same time with the large-scale, distributed, and constantly changing nature of these systems. In this paper, we introduce a distributed models@run.time approach, combining ideas from reactive programming, peer-to-peer distribution, and large-scale models@run.time. We define distributed models as observable streams of chunks that are exchanged between nodes in a peer-to-peer manner. A lazy loading strategy allows to transparently access the complete virtual model from every node, although chunks are actually distributed across nodes. Observers and automatic reloading of chunks enable a reactive programming style. We integrated our approach into the Kevoree Modeling Framework and demonstrate that it enables frequently changing, reactive distributed models that can scale to millions of elements and several thousand nodes.**

***Index Terms*—Models@run.time, Distributed models, Reactive programming, Asynchronous programming, Peer-to-peer**

I. Introduction

Over the past few years the models@run.time paradigm has proven the potential of models to be used not only at design-time but also at runtime to represent the context of cyber-physical systems (CPS), to monitor their runtime behaviour and reason about it, and to react to state changes [1], [2]. Reasoning on the state of cyber-physical systems is a complex task since it relies on the aggregation and processing of various constantly evolving data such as sensor values. The recent trend towards highly interconnected cyber-physical systems with distributed control and decision-making abilities [3], [4] makes the reasoning even more difficult. Nonetheless, to fulfill their tasks, these systems typically need to share context and state information between computational nodes (any computer system reading, writing, or processing data in the context of a CPS). Given the fact that the models@run.time paradigm promotes the use of models to represent the state and context information of cyber-physical systems, the runtime models of distributed cyber-physical systems must also be distributed. Moreover, runtime models of complex cyber-physical systems can get very large, making it very difficult to share this information efficiently.

Let us take a concrete example. We are working with Creos Luxembourg S.A., the main electricity grid operator in Luxembourg, on a smart grid project to make the electricity grid able to dynamically react and adapt itself to evolving contexts. The smart grid is characterized as a very complex and highly distributed cyber-physical system [5] where various sensor data and information from the electrical topology must be aggregated and analyzed. To support reasoning and decision-making processes we use a model of the smart grid during runtime. The main entities of this model are smart meters, data concentrators, and the underlying electric grid topology. Smart meters, installed at customers' houses, continuously measure the electric consumption and quality of power supply and steadily send this data to so-called data concentrators. Each data concentrator controls a number of associated smart meters, collects, stores, and processes the data received from these meters. The state of the smart grid, *i.e.,* its runtime model, is continuously updated with a high frequency from various sensor measurements (like consumption or quality of power supply) and other internal or external events (*e.g.,* overload warnings). In reaction to these state changes different actions can be triggered. However, reasoning and decision-making processes are not centralized but distributed over smart meters, data concentrators, and a central system [6], making it necessary to share context information between these nodes. The fact that runtime models of smart grids, depending on the size of a city or country, can reach millions of elements and thousands of distributed nodes, challenges the efficiency of sharing context information.

These challenges are not specific to the smart grid but also arise in many other large-scale, distributed cyber-physical systems where state and context information change frequently. For example advanced automotive systems, process control, environmental control, avionics, and medical systems [4].

Despite the fact that models@run.time enables the abstraction of such complex systems during runtime, to the best of our knowledge, there is no approach tackling the *i)* **large-scale**, *ii)* **distributed**, and *iii)* **constantly changing nature** of these systems at the same time [7], [8]. This paper introduces a distributed models@run.time approach combining ideas from asynchronous, reactive programming, peer-to-peer distribution, and large-scale models@run.time. First of all, since models@run.time are continuously updated during the execution of a system, they cannot be considered as bounded but can change and grow indefinitely [9]. Therefore, we define models as observable streams of model chunks, where every chunk contains data related to one model element (*e.g.,* a meter). This stream-based interpretation of models allows to process models chunk-by-chunk regardless of their global size. Secondly, we distribute and exchange these model chunks between nodes in a peer-to-peer manner and on-demand to avoid the exchange of full runtime models. Moreover, the use of a lazy loading strategy allows to transparently access the complete virtual model from every node, although chunks are actually distributed across nodes. Thirdly, we leverage observers, an automatic reloading mechanism of model chunks (in case of changes), and asynchronous operations to enable a reactive programming style, allowing a system to dynamically react to context changes.

We integrated our approach into the Kevoree Modeling Framework [10], [11] by entirely rewriting its core to apply a thoroughly reactive and asynchronous programming model. Evaluated on an industrial-scale smart grid case study, inspired by the Creos project, we demonstrate that our approach enables frequently changing, reactive distributed models and can scale to millions of elements distributed over thousands of nodes, while the distribution and model access remains fast enough to enable reactive systems.

The remainder of this paper is as follows. In Section II, we introduce the background of this work: reactive programming, peer-to-peer distribution, and the Kevoree Modeling Framework. Section III presents our approach of reactive distributed models at runtime, which we evaluate in Section IV. The related work is discussed in Section VI. In Section V we discuss the need for asynchronicity to distribute models before we conclude in Section VII.

II. Background

This section introduces important principles of reactive programming, peer-to-peer distribution, and of the Kevoree Modeling Framework.

A. Reactive Programming

Reactive programming is a paradigm focusing on observable data streams and the propagation of changes. It aims at supporting the development of asynchronous, event-driven, and interactive applications by allowing to declaratively express programs in terms of what to do when a certain event occurs [12]. Reactive programming extends the observer pattern [13] to support continuous data streams and events while abstracting from low-level tasks like threading, synchronization, and non-blocking I/O. Streams are sequences of ongoing events. Following this idea, nearly anything can be considered as a stream: sensor data, user inputs, calculation results, mouse movements and clicks, *or runtime models*. Streams can come from different sources, can be observed and reacted to. Events are always treated asynchronously, using a function that is executed when the event occurs, another function in case of errors, and a third function when the stream finishes (in case it is not a continuous stream). The observation of a stream is called subscribing, the defined functions are observers, and the stream itself is the observable. Streams have functional characteristics, such as immutability, which make it possible to use streams as input for other streams, to filter, combine, merge, or map streams to new streams. Combined, as explained here, asynchronous mechanisms and the observer pattern empower reactive programming as a powerful tool, providing a high level of abstraction for the development of event-driven and interactive applications. There exists a number of reactive programming frameworks and libraries for different programming languages, *e.g.,* the Rx (Reactive Extensions) [14] family and Scala.React [15]. With FrTime [16] and Flapjax [17] complete languages based on this paradigm have been created. Our approach of reactive distributed models@run.time is inspired by this increasingly popular programming paradigm and lifts its main ideas and concepts to the level of models.

B. Peer-to-Peer Distribution

Peer-to-peer (P2P) is a distributed computing or networking architecture, which is designed for sharing computer resources like content, storage, or CPU cycles [18]. P2P systems share these resources by direct exchange between equally privileged nodes, rather than relying on a centralized or intermediary control. Peers act as both consumers and suppliers of resources. Such architectures typically have characteristics like scalability, increased access to resources, ability to adapt to failures, and accommodate transient populations of nodes [18]. P2P networks establish a virtual overlay network on top of the actual physical network topology [19], where data is still exchanged over a TCP/IP network, but let peers communicate directly with each other via logical overlay links. Overlay networks are used for resource indexation and peer discovery and are often distinguished in terms of centralization (purely decentralized, partially decentralized, hypbrid decentralized) and structure (unstructured and structured) [18]. Gossip [20] and Gnutella [21] are examples for protocols for unstructured P2P networks. Structured P2P networks usually use some form of distributed hash tables, *e.g.,* Chord [22] or Koorde [23] to map resources to peers. Since P2P characteristics fit very well with the requirements of our distributed models@run.time, we build our distribution mechanism on top of a P2P architecture. We distribute the model itself, in a P2P manner, by splitting the model into chunks and spread them over the nodes. To keep track of which chunk is distributed on which node, we rely on standard solutions like distributed hash tables.

C. The Kevoree Modeling Framework

The Kevoree Modeling Framework (KMF) [10], [11] is an alternative to the Eclipse Modeling Framework (EMF) [24]. Like EMF, KMF is a modeling framework and code generation toolset for building object-oriented applications based on structured data models. However, while EMF was primary designed to support design-time models, KMF is specifically designed to support the models@run.time paradigm and targets runtime models. While the basic concepts remain similar, runtime models —especially runtime models of complex cyber-physical systems— usually have higher requirements regarding memory usage, runtime performance, and thread safety. Therefore, EMF faces some limitations in supporting the models@run.time paradigm, which KMF tries to address [10]. KMF also provides a notion of time [25] and a native versioning concept to support historized models [26], particularly helpful when systems have to access and correlate data from past states. KMF is able to export/import models into/from the Ecore format, considered as a *de facto* standard, to ensure the compatibility with EMF-based tools. KMF promotes the use of models not only for code generation or architectural management but also during runtime as a central artefact for the development of systems. For this reasons, we decided to integrate our approach into KMF.

III. Reactive Distributed Models@run.time

This section details our approach of reactive peer-to-peer distributed models@run.time. It begins with an overview of our proposition and defines important terms used in the rest of this paper. It then describes how models are split into chunks to allow to define models of arbitrary size as observable continuous streams of chunks. Next, this section details how these chunks together with peer-to-peer distribution techniques, lazy loading, and automatic chunk reloading are used to transparently distribute runtime models over nodes. To finish, this section presents how asynchronous programming empowers the reactivity of systems to changes and events.

A. Overview: Distributed Models as Data Stream Proxies

The goal of our contribution is to enable *i)* large-scale, *ii)* distributed, and *iii)* constantly changing models@run.time that can scale to millions of elements distributed over thousands of nodes, while keeping the distribution and model access fast enough to enable reactive systems. To address the distribution and the context sharing need, we propose a concept of runtime models, which are virtually complete and spread over the nodes of a distributed cyber-physical system. Indeed, every model element can be accessed and modified from every node, regardless on which nodes the model element is physically present. To tackle the large-scale aspect, data are never copied *a priori*. Instead runtime models are considered as proxies of data, loading the related data only on-demand. This is achieved by splitting runtime models into streams of data chunks, where every chunk corresponds to one model element. These data chunks are physically distributed in a peer-to-peer manner, using distribution strategies similar to those used for media sharing. Finally, reactive programming concepts and a fine-grain (*i.e.,* per model element) load and update strategy are used together to cope with the constantly changing nature of model elements. Asynchronous operations allow to address the inherent uncertainty of network communications and the reactive aspect empowers models to dynamically react by observing changes on the shared data stream. These characteristics are closely interlinked and we claim that the combination of the three can offer distributed and scalable models@run.time able to deal with constantly changing model elements. Our approach is depicted in Figure 1 and detailed in the rest of this section.

B. Definition of Terms

Following model-driven engineering concepts, we generate an object-oriented API based on a meta model. This API can be used to instantiate and manipulate a runtime model. This is depicted on the top of Figure 1. The runtime model conforms to the meta model and is used during the execution of a system. In our case, we use KMF to generate a Java API to create and manipulate runtime models, which can be used during the execution of a cyber-physical system to represent its context, to monitor its runtime behaviour, and to react to state changes. In the following we use the term *model* as a synonym for *runtime model*. In cases where we refer to a *meta model* we explicitly use the term *meta model*. We use the term *model element* to refer to one object of the runtime model corresponding to one meta class in the meta model. A *chunk* is the (serialized) content of one model element. This can be seen in Figure 1.

C. Models@run.time as Streams

In a formal way, we denote by N the set of the connected nodes. Every node $n_i \in N$, is a tuple of unique id_i and an (infinite) sequence of model chunks S_{n_i}. $S_{n_i} = \{s_{i1}, \ldots, s_{ij}, \ldots\}$. Each model chunk s_{ij} contains an atomic update for one runtime model element me_k. A runtime model element has a unique $uuid$ [27], which is automatically generated when a model element is created during runtime. This model element contains a set of attributes and a set of relationships to other model elements. Our global distributed runtime model M, is thus the aggregation of all these distributed model elements. As argued, runtime models of cyber-physical systems need to be continuously updated to reflect state changes in the underlying systems. This makes it difficult to consider them as bounded [9]. Moreover, as we have already defined a one-to-one mapping between the model element updates and the model chunks s_{ij}, we can consider that M is the virtual aggregation of all the model chunks created on all the nodes: $M = \cup S_{n_i}, \forall n_i \in N$. This is described in Figure 1.

Next, we define two functions, $serialize$ and $unserialize$, for every model element. The function $serialize$ takes a model element as input and creates a compact JSON format *(similar to BSON)* of the runtime model element as output. It contains the $uuid$ and data of the model element. The $uuid$ of a model element is immutable, meaning that it cannot be changed during the whole lifetime of an element. For efficiency reasons we use a $long$ value for $uuid$s. Similarly, the $unserialize$ function takes a JSON representation of the model element as input and creates a model element as output. The $uuid$ of

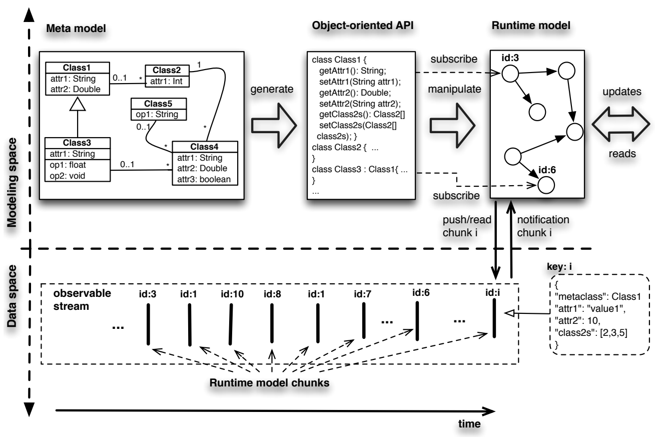

Fig. 1. Models as continuous streams of chunks

model elements together with the $serialize$ and $unserialize$ functions allow us to split models in a number of chunks.

It is important to note that for each point in time a model still consists of a finite number of chunks. However, considering the fact that a model can continuously, *i.e.,* infinitely, evolve over time a model can be interpreted as an infinite stream of model chunks, where every model element changed is added to the stream. Newly created model elements are considered in the same way as changes. To delete elements we define an explicit function $delete$. This function removes an element from all relations where it is referenced. In addition, all elements contained in the deleted one are recursively considered as deleted. Streams are naturally ordered by time, *e.g.,* based on a clock strategy [28].

The definition of a $uuid$ for every model element and the way we split models into chunks also allow us to leverage a lazy loading [29] strategy for chunks. Indeed, references are resolved on-demand, meaning that the actual data of model elements are loaded only when accessed by an explicit request or by traversing the object model. For *one-to-one* relationships, the chunks only contain the $uuid$ of the target model element, and a (primitive) array of $uuid$s in case of *one-to-many* relationships. If a relationship changes the chunk is updated. An example of a chunk can be seen in the lower right corner of Figure 1. With this strategy we enable the loading of model elements on-demand, regardless if they come from a file, local or remote database, or —as discussed in subsection III-D— if they are distributed in a peer-to-peer manner. Since model element chunks can potentially be modified from a concurrent (local or remote) process, we reload model element chunks when they are accessed again. To reduce the overhead caused by this reloading, we use a caching strategy [30] to decide wether an element needs to be reloaded or not. This is a trade-off between the freshness of data and overhead of reloading.

In Section IV, we demonstrate that this approach enables to efficiently process large-scale models that do not necessarily fit completely into main memory, by allowing to manipulate models chunk-by-chunk. In addition, this lays the foundation for our peer-to-peer-based distribution approach.

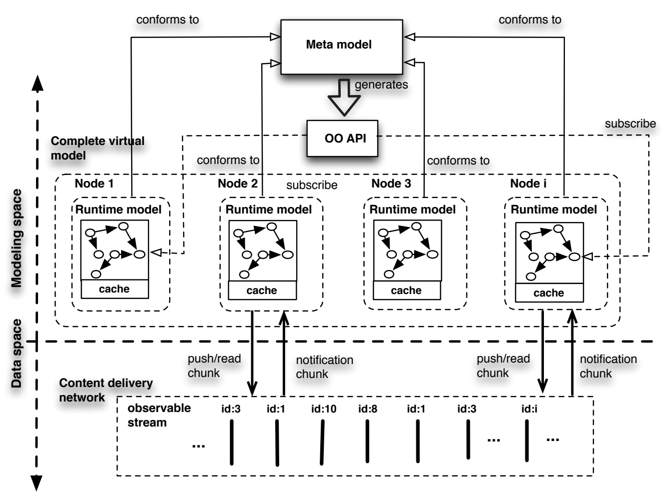

Fig. 2. Distribution model

D. Distributed Models@run.time

This subsection describes the peer-to-peer distribution of data chunks. We first discuss how we enable distributed modeling and how we uniquely identify distributed model elements. Then, we outline the generic content delivery network (CDN) interface to bridge the gap between the model and data space. Finally, we discuss how we distribute data chunks using distributed hash tables (DHT).

The idea of our contribution is to offer a virtually complete view of runtime models, even though they are actually distributed over several nodes. To ensure consistency between runtime models they all conform to the same meta model. In future work we want to investigate how an independent evolution of meta models could be achieved, *e.g.,* by (semi) automatically migrating chunks on the fly. This would enable different nodes to use different versions of a meta model. To avoid the need of a-priori replication, we use runtime models as data proxies. The task of model proxies is to decompose all model operations into data chunk operations. This is the responsibility of a so-called content delivery network, which provides operations to retrieve *(get)* and share *(put)* data chunks. Figure 2 depicts the distribution architecture of our approach.

Since model elements can be created on any node, we first have to refine our $uuid$ generation strategy to avoid $uuid$ overlaps. Consensus algorithms like RAFT [31] or Paxos [32] are able to offer strong consistency guaranties. However, they are not designed for very high volatility like needed for object creation. Therefore, we define our $uuid$s with the goal to reduce the amount of consensus requests based on a leader approach [31]. We use 64 bits $uuid$s composed of two parts. A 20 bits $prefix$ (MSB: most significant bits) is negotiated at the time a node connects to the CDN and is from this point assigned to the node. The remaining 44 bits are used for locally generated identifiers (LSB: least significant bits). This composition of $uuids$ is depicted in Figure 3.

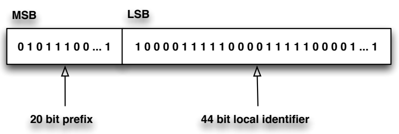

Fig. 3. Composition of uuids for distributed models

$Prefix$es are allocated in a token ring and can therefore be reused. The number of concurrent connections is limited by the $prefix$ size. With 20 bits $prefix$es we enable more than one million connections without a risk of collision. Managing $prefixes$ in a token ring and reusing them allow us to use more than one million connections, but with an increasing number of connections the risk of collisions also increases. By using 44 bit for local identifiers every node can create 2^{44} objects per $prefix$ (about $17,592$ billions). If a node needs to create more objects, another $prefix$ must be requested from the CDN. As depicted in Figure 2, every node relies on a content delivery network, which is responsible for the data chunk exchange strategy, $prefix$ negotiation, and network related operations. Different implementations of the CDN can support different distribution scenarios. We now focus on peer-to-peer distribution. Listing 1 illustrates a simplified interface definition of a CDN driver.

Listing 1. CDN interface
```
interface ModelContentDelveryDriver {
  atomicGet(byte[] key, Callback<byte[]> callback);
  get(byte[] key, Callback<byte[]> callback);
  put(byte[] key, byte[] val, Callback callback);
  multicast(long[] nodes,byte[] val, Callback callback);
}
```

The method $atomicGet$ is used during the $prefix$ negotiation phase and requires a consensus like algorithm, *e.g.,* RAFT. The methods get and put are the two primary methods to load, store, and share data chunks. These operations can be implemented using a multicast dissemination strategy or with more advanced concepts like distributed hash-table algorithms. These algorithms offer partitioning of storage to replicate data over network. Finally, the method $multicast$ is used for the dissemination of modification events to all subscribed nodes. All these methods are asynchronous and use callbacks to inform about their execution results.

A CDN driver needs to be instantiated at each node to enable nodes to collaborate to exchange data chunks. Like the PeerCDN project [33], the CDN implementation used in our approach relies on the Kademlia distributed hash table to implement the get and put operations. Since Kademlia scales very well with the number of participants, we leverage it on top of a WebSocket communication layer.

E. Reactive Models@run.time

As discussed, a key requirement for models@run.time-based systems is to be able to quickly react to state changes. In order to ensure reactivity, we make our streams of model chunks observable [13]. We enable runtime models to subscribe to these observable streams. Therefore, we define an API that allows to specify which model elements should be observed. This is usually domain-specific knowledge. Then, whenever one of these runtime model elements changes (regardless if due to local or remote changes) the observer (a runtime model) is notified. Different runtime models can observe different model elements, depending on which changes are important for this observer. For example, the top of Figure 1 shows that the runtime model subscribes to changes of the two runtime model elements with $id = 3$ and $id = 6$. This means that whenever one of the model elements with $id = 3$ or $id = 6$ changes, the observer (runtime model) is notified. The information, which runtime model observes which model elements, is managed by the CDN and stored in a distributed hash table. Listing 2 shows how the generated API can be used to subscribe for model element changes.

Listing 2. Subscription for model changes
```
runtimeModel.subscribeAll(false);
runtimeModel.subscribe(3, new Callback() {
    /* callback code */ };
runtimeModel.subscribe(6, new Callback() {
    /* callback code */ };
);
```

There are two important concepts to note. First, explicitly subscribing only to elements that are important for the computational node hosting this runtime model reduces the unnecessary propagation of information through the network. Secondly, it can be specified what should happen, *i.e.,* what code should be executed, when the observed change occurs. This allows a reactive programming style by declaratively specifying what should happen if a certain event occurs. As can be seen in Listing 2, the second parameter of the $subscribe$ method is a callback, meaning that this is a non-blocking code. Since distributed systems are inherently asynchronous this non-blocking capability is key [34]. Without the support of non-blocking operations this would mean that a computation node is blocked until the awaited event occurs. For example, if a data concentrator intends to read the consumption value of an associated smart meter, with a blocking operation the concentrator (thread) would be blocked until the value arrives. Since this can take several seconds the computation time of the concentrator is wasted and cannot be used for something else in the meantime. Blocking operations are therefore contradictory to the requirement that models@run.time-based systems need to be able to quickly react to state changes.

Current standard modeling frameworks like EMF are strictly synchronous. This makes them inappropriate for highly distributed and asynchronous applications. Thus, we completely rewrote the core of KMF to apply a thoroughly reactive and asynchronous programming model. In fact, every method call in KMF is asynchronous and therefore non-blocking. This principle is shown on the right side of Figure 4. The left side of Figure 4 shows in comparison a synchronous, *i.e.,* blocking operation call. As can be seen in the figure, non-blocking operation calls do not block the calling process until the triggered operation is finished. Therefore, the caller

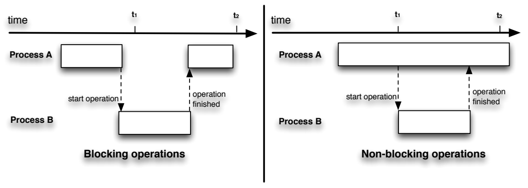

Fig. 4. Blocking and non-blocking operation calls

process can do something else in the meantime. As soon as the operation execution finishes, the caller is notified using a callback. To avoid deeply nested callbacks, which is occasionally referred to as *callback hell* [35], every method call in KMF immediately returns a *KDefer* object, which is similar to a *Future* or *Promise*. These objects enable the definition of a control flow by specifying that the execution of one *KDefer* depends on the result of another *KDefer* and so on. Listing 3 shows how this looks like.

Listing 3. Asynchronous method calls with KDefer
```
KDefer filter = runtimeModel.createDefer();
KDefer defer = class2.getClass2s();
filter.wait(defer);
filter.setJob(new KJob() { /* filter class2s */ });
```

Listing 3 filters the results of the getter of $class2s$. However, the getter is asynchronous and therefore filtering the result can only start when the getter is executed. This is realized by defining that the filter has to wait for the results of the getter. It is important to note that this is not an active wait. Instead, the control flow immediately continues (non-blocking) and the execution of the filter object is delayed until the getter finishes. To make it easier to traverse models (without the need to deal with callbacks) we define a traversal language on top of KMF. The execution of *KDefer*s are transparently mapped to processes in KMF. A task scheduler system allows to specify a specific strategy. This allows a MapReduce-like [36] approach to horizontally scale by parallelizing method calls.

IV. Evaluation

In this section we evaluate our reactive peer-to-peer distributed models@run.time approach. We show that it can scale to runtime models with millions of elements distributed over thousands of nodes, while the distribution and model access remain fast enough to react in near real-time. Our evaluation is based on a smart grid case study, inspired from a real-world smart grid project. We implemented and integrated our approach into the Kevoree Modeling Framework. We first introduce the smart grid case study before we evaluate our approach and present the results.

A. The Smart Grid Case Study

We work together with our industrial partner Creos Luxembourg S.A. to make the electricity grid able to dynamically react and adapt itself to evolving contexts. For the context of this project we define a model of the main grid devices and use it during runtime for monitoring and reasoning purposes.

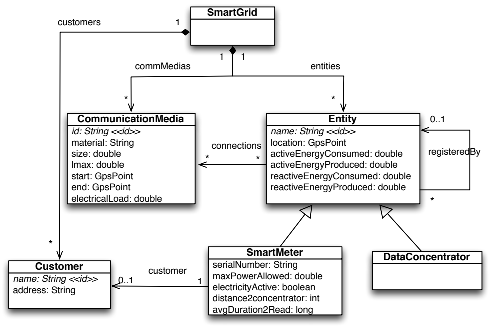

Fig. 5. Smart grid meta model

The main devices of the smart grid infrastructure are smart meters and data concentrators [37]. Smart meters, installed at customers' houses continuously measure electric consumption, quality of power supply and support load management. In regular intervals (*e.g., every 15 minutes*), smart meters report the consumption data to their associated concentrators. Data concentrators collect and store consumption data from a number of associated meters. Concentrators are able to send commands to these smart meters, *e.g.,* requesting consumption data, restricting the maximum load, or shutting down the electricity of a customer. Concentrators divide the smart grid topology into smaller regions and enables what is referred to as distributed control ability [6]. This also leads to a distribution of data and processing. However, reasoning and decision-making processes, *e.g.,* electric load approximation, usually need to aggregate and process data from several concentrators. Therefore, we need to efficiently distribute our runtime models over several concentrators but still be able to transparently access all data from every node. Figure 5 shows a simplified (stripped-down to focus on the essentials of the contribution of this paper) excerpt of the meta model that we use in this project. This meta model is also used for our evaluation.

B. Evaluation Setting

We evaluate our approach in terms of its capability to tackle our three main requirements: *i)* large-scale models, *ii)* distributed models, and *iii)* frequently changing models. We use the smart grid model presented in IV-A and vary it in size and distribution. Experiments are executed on a 2,6 GHz Core i7 CPU with 16GB RAM and SSD drive using the Java version of KMF. We use Docker *(http://www.docker.com/)* containers to distribute nodes. Every presented value is averaged from 10 measurements. The evaluation experiments are available on GitHub *(https://github.com/kevoree/xp-models15/).*

C. Scalability for Large-Scale Models

In this benchmark we investigate the scalability characteristics of our approach for large-scale models. Therefore, we read a number of model elements and analyze how the performance of this is impacted by the model size. As discussed, complex cyber-physical systems often need to leverage very large models for their reasoning tasks. However, many operations performed on shared models only use a small fraction of the complete model for their reasoning activities. As argued in this paper, this is often due to the distributed nature of processing tasks. For instance, in the smart grid scenario, every concentrator mainly needs the part of the model representing its district. Therefore, we evaluate the performance of reading a constant number of model elements (25 elements) and analyze how this is impacted by the model size. In this experiment, we increase the number of model elements step by step to more than 1.5 million elements, while the model is distributed over two nodes. In the next subsection we analyze the effect of a highly distributed model. For each model size, we read 25 elements from the model. More specifically, we read consumption values of smart meters in order to approximate the electric loading. The read operations are performed on one node, which communicates through a WebSocket communication protocol with the other node. Since models are composed of object graphs, the performance of read operations usually differs depending if a model is very deep or wide. For this reason, we evaluated both scenarios: once we increased the model size in depth and once in width. Our results are presented in Figure 6. It is important to note

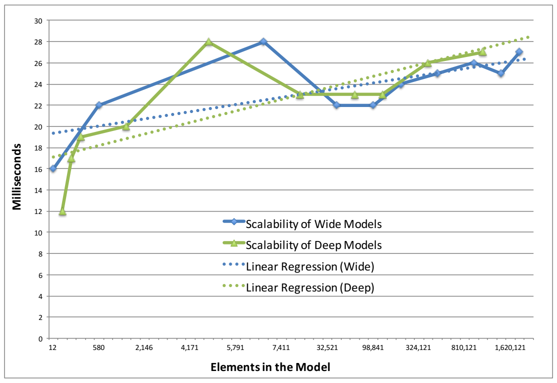

Fig. 6. Scalability of read operations for large-scale models

that the time to load the model elements is barely affected by the model size. In fact, scalability for models which are large in width, is nearly constant while for models which are large in depth is nearly linear. In this experiment, we demonstrated that our distributed models@run.time approach allows distributed read operations in an order of magnitude of milliseconds (*between 12 and 28 ms*) in a model with millions of elements. From this experiment, we can conclude that our concept of model elements, which act as proxies on a stream of data chunks, is suitable for large-scale models@run.time.

D. Scalability for Large-Scale Distribution

In this experiment we investigate the ability of our distributed runtime model approach to collaborate with a huge number of nodes through a shared common context. We evaluate the capacity of the model to propagate changes to a huge amount of collaborating nodes. This large scale distribution is representative for smart grid architectures. To conduct the experiments we used five physical computers (Intel Core i7 with 16GB RAM), connected through a local area network. On each computer we sequentially started 200 virtual docker nodes using the Decking tool *(http://decking.io)* (from 200 up to 1000 nodes) and measured the propagation time of model updates. The 200 docker nodes per physical computer result from a limit of the Linux Kernel. We simulate changes in the smart grid topology which have to be propagated to all nodes. For this, every five seconds one of the nodes (virtual machines) in the network is updating a value and propagates this change. We measure the required time for propagating the changes. Figure 7 shows the results in five scales, represented by the probability spectral density (grouped by 10 ms), reaching from **200** to **1000** collaborating nodes. The spectral density reflects

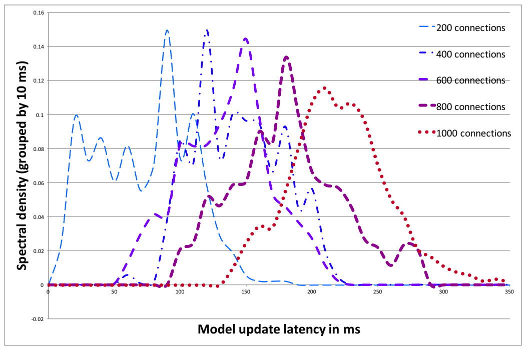

Fig. 7. Spectral probability density of the model update latency

the probability of each latency depending on the number of nodes. With this benchmark, we demonstrate that our approach can handle a high number of collaborating nodes while the latency remains low. The raw results are shown in Table I.

Nodes Nb.	Min(ms)	Max(ms)	Avg(ms)
200	11	188	88.01
400	63	220	128.75
600	87	253	169.52
800	102	289	185.62
1000	141	355	224.66

TABLE I
MEASURED LATENCY (IN MS) TO PROPAGATE CHANGES

E. Scalability for Frequently Changing Models

Cyber-physical systems and their associated sensors lead to frequent updates in their associated context models. Therefore, in this benchmark we evaluate the ability of our distributed data stream concept to partly update runtime models with a high frequency. In the following benchmarks we use two nodes connected through a WebSocket connection. In a first benchmark, we investigate the highest possible volatility of a model element. On a model with 1.5 million elements, we evaluate how frequently a single element can be updated per second. We evaluated the time to update the value of an attribute on one node and to send an update notification to the node. We measured the maximal frequency our implementation is able to handle: **998 updates per second** for a model with 1.5 million elements.

In a second benchmark we evaluated the ability of our approach to handle changes of different size in large models. Therefore, we first updated a small percentage of the model and then increased the percentage of changes step by step. We measured the time needed to update the model and inform the other node about the change (context sharing). Figure 8 presents our results. The results show that our approach

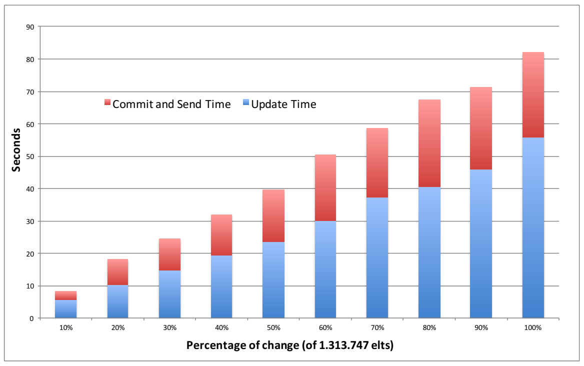

Fig. 8. Required time for update operations of different size

approximately scales linear to the percentage of changes. For changes of a small part of the model *(around 10% which is equivalent to 150000 elements)* our approach remains below 10 seconds. Only if 70% or more of the model changes the update and propagation time exceeds one minute. These results show that our approach is able to handle a high volatility of model elements while still offering good latency properties.

V. Discussion: Distribution and Asynchronicity

The border between large-scale data management systems and models is becoming more and more fuzzy as models@run.time progressively gains maturity through large-scale and distribution mechanisms. Therefore it is clearly important to evaluate the limit and the potential reuse of each domain. For instance, despite the feasibility of mapping a models@run.time into a distributed database, which takes care of the replication of data, it quickly leads to many limitations in practice. Indeed, to mimic synchronous calls, heavy and costly distributed algorithms have to be involved, such as consensus or RAFT. However, because every communication can fail, the uncertainty is at the heart of the distribution. Instead of hiding it, most of nowadays software stacks exploit explicit asynchronous programming to scale their distributed computation. In this trend we can mention the well-known AJAX, or even the API of NodeJS server or distributed P2P communication, which are by default asynchronous. Therefore, beyond the ability to distribute in a scalable manner models@run.time over nodes, our contribution also includes at its core an asynchronous layer in models. We are convinced that this change is inescapable if runtime models want to go beyond the barrier of computer memory in order to exploit the power of distributed systems. Moreover, asynchronous modeling pave the way clearly to define the semantic of partial and large scale models.

VI. Related Work

The closest to the presented contribution is the work of Fouquet *et al.,* [38]. They discuss the challenge how to propagate reconfiguration policies of component-based systems to ensure consistency of architecture configuration models over dynamic and distributed systems. They propose a peer-to-peer distributed dissemination algorithm for the models@run.time context based on gossip algorithms and vector clock techniques that are able to propagate the reconfiguration policies. While their goal is essentially to propagate changes made in the model of one computation node to the model of other computation nodes, their approach differs significantly from ours. First, they focus mainly on architectural configuration models [39], which are typically of manageable size and can be exchanged in one piece. On the contrary, we focus on big runtime models supporting millions of elements over several thousand distributed instances, making it basically impossible to exchange the complete model in reasonable time. Secondly, our approach promotes observable streams and asynchronous operations enabling a reactive programming style. Last but not least, instead of using a gossip and vector clock-based dissemination strategy to ensure model consistency, we rely on protocols like web sockets or WebRTC together with lazy loading to stream our models between distributed nodes.

Several authors identified the need of infinite, unbounded models and some sort of model streaming. In [9] Combemale *et al.,* propose a formal definition of an infinite model as an extension of the MOF formalism together with a formal framework to reason on queries over these infinite models. Their work aims at supporting the design and verification of operations that manipulate infinite models. Particularly, they propose formal extensions of the MOF $upperbound$ attribute of the $Property$ element to define infinite, unbounded collections and iterate over them. For similar reasons, we also define models as infinite streams of model chunks. However, our approach goes beyond this and allows the distribution of model chunks over nodes and the definition of repeatable asynchronous operations to lazily (re)load these chunks from remote nodes. This is somewhat similar to what is proposed for stream processing [40] in programming languages or database management systems.

In [41] Cuadrado *et al.,* discuss the motivation, scenarios, challenges, and initial solutions for streaming model transformations. They motivate the need for this new kind of transformation with the fact that a source model might not be completely available at the beginning of the transformation, but might be generated step by step. They present an approach and provide a prototype implementation, built on top of the Electic transformation tool. In a similar direction goes the work of Dávid *et al.,* [42]. They suggest to use incremental model query techniques together with complex event processing to stream model transformations. Their approach foresees to populate event streams from elementary model changes with an incremental query engine and to use a complex event processing engine to trigger transformation rules. This is applied for gesture recognition for data coming from a KINECT sensor. On the contrary to [41] their approach uses derived information regarding the model in the form of change events, which decouples the execution from the actual model. Ráth [43] presents an approach for change-driven model transformations, directly triggered by complex model changes carried out by arbitrary transactions on the model. They identify challenges for change-driven transformations and define a language for specifying change-driven transformations as an extension of graph patterns and graph transformation rules. Our and these approaches have in common that we identify a need for continuous or infinite models. Unlike these approaches we do not stream events or model transformations and use complex event processing engines to detect complex events, but view runtime models itself as continuous streams.

Several authors worked on the issues of large-scale models, model persistence, and the fact that they might grow too big to fit completely into main memory. Pagán *et al.,* [29] propose Morsa, an approach for scalable access to large models through on demand loading. They suggest to use NoSQL databases for model persistence and provide a prototype that integrates transparently with EMF. In their evaluation they showed that they have significantly better results than the EMF XMI file-based persistence and CDO. In a similar direction goes the work of Koegel and Helming [44], Gomez *et al.,* [45], or Hartmann *et al.,* [26]. However, none of this work addresses distribution or asynchronicity. To address the scalability of queries, Szárnyas *et al.,* [46] present an adaption of incremental graph search techniques, like EMF-IncQuery. They propose an architecture for distributed and incremental queries.

VII. Conclusion and Future Work

Cyber-physical systems, such as smart grids, are becoming more and more complex and distributed. Despite the fact that models@run.time enable the abstraction of such complex systems during runtime and to reason about it, the combination of the *i)* large-scale, *ii)* distributed, and *iii)* constantly changing nature of these systems is a big challenge. These characteristics are closely interlinked: The increasing complexity of cyber-physical systems naturally leads to bigger models, which are —due to their size— also more difficult to distribute or replicate. Finally, the distributed aspect inherently leads to asynchronicity and this in turn requires the ability to dynamically react to events instead of actively waiting. Therefore this paper introduces a distributed models@run.time approach, combining ideas from reactive programming, peer-to-peer distribution, and large-scale models@run.time. First, since models@run.time are continuously updated during the execution of a system, they cannot be considered as bounded but can change and grow indefinitely. We defined models as observable streams of model chunks, where every chunk contains data related to one model element. This stream-based interpretation of models allows to process models chunk by chunk regardless of their global size. Secondly, we distribute and exchange these model chunks between nodes in a peer-to-peer manner and on-demand to avoid the necessity to exchange full runtime models. Nonetheless, the use of a lazy loading strategy allows to transparently access the complete virtual model from every node, although chunks are actually distributed across nodes. Thirdly, we leverage observers, an automatic reloading mechanism of model chunks (in case of changes), and asynchronous operations to enable a reactive programming style, allowing a system to dynamically react to context changes. We integrated our approach into the Kevoree Modeling Framework and evaluated on an industrial-scale smart grid case study. We demonstrated that this approach can enable frequently changing, reactive distributed models and can scale to millions of elements distributed over thousands of nodes, while the distribution and model access remains fast enough to enable reactive systems.

In future work we plan to extend our approach to classical client-server applications, enabling a client and server to operate on the same runtime model. All model changes made in either the server or the client would be automatically synchronized to the other one by considering the runtime model as a distributed model. This would make the typical data transform-send-transform process unnecessary, which is in many of today's architectures, like SOA, mandatory. The reactive aspect of our approach would allow clients to dynamically react to changes by updating the corresponding view. For example, if a client shows stock prices and a value is changed on the server, this change would be fully automatically and transparently propagated to the client, which can react to it. By using isomorphic models (models of different languages which use the same API) this could also be used if client and server applications are written in different languages, *e.g.,* Java und JavaScript. Another goal for a future work is to enable the evolution of meta models, *i.e.,* to enable different nodes to use different versions of a model. This is currently not supported in our approach. We rely on the fact that every node uses runtime models conforming to the same meta model version.

Acknowledgment

The research leading to this publication is supported by the National Research Fund Luxembourg (grant 6816126 and CoPAInS project CO11/IS/1239572) and Creos Luxembourg S.A. under the SnT-Creos partnership program.

References

[1] G. S. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[2] N. Bencomo, R. B. France, B. H. C. Cheng, and U. Aßmann, Eds., *Models@run.time - Foundations, Applications, and Roadmaps [Dagstuhl Seminar 11481, November 27 - December 2, 2011]*, ser. Lecture Notes in Computer Science, vol. 8378. Springer, 2014.

[3] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, June 2010, pp. 731–736.

[4] E. A. Lee, "Cyber physical systems: Design challenges," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*. IEEE, 2008, pp. 363–369.

[5] J. Taneja, R. Katz, and D. Culler, "Defining cps challenges in a sustainable electricity grid," in *Proceedings of the 2012 IEEE/ACM Third International Conference on Cyber-Physical Systems*. IEEE Computer Society, 2012, pp. 119–128.

[6] H. Farhangi, "The path of the smart grid," *Power and Energy Magazine, IEEE*, vol. 8, no. 1, pp. 18–28, 2010.

[7] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel, "A dynamic component model for cyber physical systems," in *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*. ACM, 2012, pp. 135–144.

[8] A. Taherkordi, F. Loiret, R. Rouvoy, and F. Eliassen, "Optimizing sensor network reprogramming via in situ reconfigurable components," *ACM Trans. Sen. Netw.*, vol. 9, no. 2, pp. 14:1–14:33, Apr. 2013. [Online]. Available: http://doi.acm.org/10.1145/2422966.2422971

[9] B. Combemale, X. Thirioux, and B. Baudry, "Formally defining and iterating infinite models," in *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, September 30-October 5, 2012. Proceedings*, 2012, pp. 119–133.

[10] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J. Jézéquel, "An eclipse modelling framework alternative to meet the models@runtime requirements," in *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, September 30-October 5, 2012. Proceedings*, 2012, pp. 87–101.

[11] F. Francois, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jézéquel, "Kevoree modeling framework (kmf): Efficient modeling techniques for runtime use," *arXiv preprint arXiv:1405.6817*, 2014.

[12] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, pp. 52:1–52:34, Aug. 2013.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[14] E. Meijer, "Your mouse is a database," *Queue*, vol. 10, no. 3, pp. 20:20–20:33, Mar. 2012.

[15] I. Maier, T. Rompf, and M. Odersky, "Deprecating the observer pattern," Tech. Rep., 2010.

[16] G. H. Cooper and S. Krishnamurthi, "Embedding dynamic dataflow in a call-by-value language," in *Proceedings of the 15th European Conference on Programming Languages and Systems*, ser. ESOP'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 294–308.

[17] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: A programming language for ajax applications," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. ACM, 2009, pp. 1–20.

[18] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 335–371, Dec. 2004. [Online]. Available: http://doi.acm.org/10.1145/1041680.1041681

[19] M. Kamel, C. Scoglio, and T. Easton, "Optimal topology design for overlay networks," in *NETWORKING 2007. Ad Hoc and Sensor Networks, Wireless Networks, Next Generation Internet*. Springer, 2007, pp. 714–725.

[20] Z. J. Haas, J. Y. Halpern, and L. Li, "Gossip-based ad hoc routing," *IEEE/ACM Trans. Netw.*, vol. 14, no. 3, pp. 479–491, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1109/TNET.2006.876186

[21] R. Matei, A. Iamnitchi, and I. Foster, "Mapping the gnutella network," *Internet Computing, IEEE*, vol. 6, no. 1, pp. 50–57, Jan 2002.

[22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160. [Online]. Available: http://doi.acm.org/10.1145/383059.383071

[23] M. Kaashoek and D. Karger, "Koorde: A simple degree-optimal distributed hash table," in *Peer-to-Peer Systems II*, ser. Lecture Notes in Computer Science, M. Kaashoek and I. Stoica, Eds. Springer Berlin Heidelberg, 2003, vol. 2735, pp. 98–107.

[24] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[25] T. Hartmann, F. Fouquet, G. Nain, B. Morin, J. Klein, and Y. L. Traon, "Reasoning at runtime using time-distorted contexts: A models@run.time based approach," in *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013.*, 2014, pp. 586–591.

[26] T. Hartmann, F. Fouquet, G. Nain, B. Morin, J. Klein, O. Barais, and Y. L. Traon, "A native versioning concept to support historized models at runtime," in *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, 2014, pp. 252–268.

[27] D. Kolovos, D. Di Ruscio, A. Pierantonio, and R. Paige, "Different models for model matching: An analysis of approaches to support model differencing," in *Comparison and Versioning of Software Models, 2009. CVSM '09. ICSE Workshop on*, May 2009, pp. 1–6.

[28] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[29] J. E. Pagán, J. S. Cuadrado, and J. G. Molina, "Morsa: A scalable approach for persisting and accessing large models," in *Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 77–92.

[30] F. Khunjush and N. J. Dimopoulos, "Lazy direct-to-cache transfer during receive operations in a message passing environment," in *Proceedings of the 3rd Conference on Computing Frontiers*, ser. CF '06. New York, NY, USA: ACM, 2006, pp. 331–340.

[31] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annual Tech. Conf.*, 2014, pp. 305–320.

[32] P. Dutta, R. Guerraoui, and L. Lamport, "How fast can eventual synchrony lead to consensus?" in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. Int. Conf. on*. IEEE, 2005, pp. 22–27.

[33] J. Wu, Z. Lu, B. Liu, and S. Zhang, "Peercdn: A novel p2p network assisted streaming content delivery network scheme," in *Computer and Information Technology, 2008. CIT 2008. 8th IEEE International Conference on*. IEEE, 2008, pp. 601–606.

[34] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[35] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones, "Calling hell from heaven and heaven from hell," in *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '99. New York, NY, USA: ACM, 1999, pp. 114–125. [Online]. Available: http://doi.acm.org/10.1145/317636.317790

[36] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[37] T. Hartmann, F. Fouquet, J. Klein, Y. L. Traon, A. Pelov, L. Toutain, and T. Ropitault, "Generating realistic smart grid communication topologies based on real-data," in *2014 IEEE International Conference on Smart Grid Communications, SmartGridComm 2014, Venice, Italy, November 3-6, 2014*, 2014, pp. 428–433.

[38] F. Fouquet, E. Daubert, N. Plouzeau, O. Barais, J. Bourcier, and J. Jézéquel, "Dissemination of reconfiguration policies on mesh networks," in *Distributed Applications and Interoperable Systems - 12th IFIP WG 6.1 International Conference, DAIS 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, 2012, pp. 16–30.

[39] B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg, "Models@ run.time to support dynamic adaptation," *Computer*, vol. 42, no. 10, pp. 44–51, Oct 2009.

[40] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '02. New York, NY, USA: ACM, 2002, pp. 1–16.

[41] J. S. Cuadrado and J. de Lara, "Streaming model transformations: Scenarios, challenges and initial solutions," in *Theory and Practice of Model Transformations*. Springer, 2013, pp. 1–16.

[42] I. Dávid, I. Ráth, and D. Varró, "Streaming model transformations by complex event processing," in *Model-Driven Engineering Languages and Systems*. Springer, 2014, pp. 68–83.

[43] I. Ráth, G. Varró, and D. Varró, "Change-driven model transformations," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, A. Schürr and B. Selic, Eds. Springer Berlin Heidelberg, 2009, vol. 5795, pp. 342–356.

[44] M. Koegel and J. Helming, "Emfstore: A model repository for emf models," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 307–308.

[45] A. Gomez, M. Tisi, G. Sunyé, and J. Cabot, "Map-based transparent persistence for very large models," in *Fundamental Approaches to Software Engineering*. Springer, 2015, pp. 19–34.

[46] G. Szárnyas, B. Izsó, I. Ráth, D. Harmath, G. Bergmann, and D. Varró, "Incquery-d: A distributed incremental model query framework in the cloud," in *Model-Driven Engineering Languages and Systems*. Springer International Publishing, 2014, pp. 653–669.