Distributed Management Information Models

Conference Paper · May 2017

DOI: 10.23919/INM.2017.7987306

3 authors:

Liam Fallon
Ericsson
44 PUBLICATIONS **99** CITATIONS
SEE PROFILE

John Keeney
LM Ericsson Ireland
81 PUBLICATIONS **658** CITATIONS
SEE PROFILE

Sven van der Meer
L.M. Ericsson
116 PUBLICATIONS **965** CITATIONS
SEE PROFILE

Distributed Management Information Models

Liam Fallon, John Keeney and Sven van der Meer
Network Management Lab, Ericsson, Athlone, Co. Westmeath, Ireland
{liam.fallon john.keeney sven.van.der.meer}@ericsson.com

***Abstract*—The use of information models to share and allow modification of network element state is one of the best and most widely adopted ideas in network management. The formal structure of information models and the controlled manner of accessing and changing such models brings both flexibility and control when managing network elements. However, keeping information models synchronized and consistent across network elements and management systems is also one of the most challenging tasks in network management system development. Today this problem is exasperated with the advent of ephemeral network functions and elements and also by the need for distributed scalable cooperating management functions running in containerized distributed cloud deployments.**

In computer science, there have been major advances in systems that allow seamless distribution of data across distributed executing entities, and separately in systems that allow highly granular data access synchronization across distributed entities. However, such systems do not place importance on "information model" concepts, with data usually distributed as largely unmodeled unstructured data maps.

In this paper, we describe our novel approach for distributed information models. We describe how information models are distributed to dispersed network elements and management systems, how synchronized access to distributed information models is achieved, how information models are persisted, and how lookups and changes to information models are logged.

I. Introduction

With the advent of cloud computing, virtualization, and hosts with multiple cores and hundreds of GB of memory, telecommunication networks are becoming increasingly distributed and virtualized. Technologies such as Software Defined Networking (SDN) and Network Function Virtualization (NFV) enable distributed virtualized "soft" networks, therefore network management applications must also evolve to manage such networks. Although specifications for managing NFV such as the NFV-MANO [1] do not prescribe distributed management, using a distributed approach in management systems is an obvious way to leverage the power of today's cloud infrastructure with its underlying multi-core and memory rich hosts. Therefore it is not surprising that modern management systems such as Ericsson's ENM [2] are inherently distributed.

Network management has long used information models to describe information that is being managed [3][4][5] in network elements and management systems and the value of having such structured models to constrain the managed information is widely recognized. Such models provide many advantages: the modeled information is unambiguously described; the model acts as an interface to the managed domain; the protocol for accessing the model is available, and any application can use and manipulate the information in the model once it complies with the protocol; and usage of the model can be logged. The paradigm of hierarchical models is ubiquitous in network management [6], with managers in a given level of management using models published by agents in the level below. However, current management information models are defined with the scope of a single Network Element (NE) or host, so co-ordination of models across multiple hosts is left as a task for each management application. The lack of common distributed management of information models is a serious drawback for distributed management application development. Not only does each application have to manage its own distributed information, there is no common way to coordinate when different distributed management applications share management information.

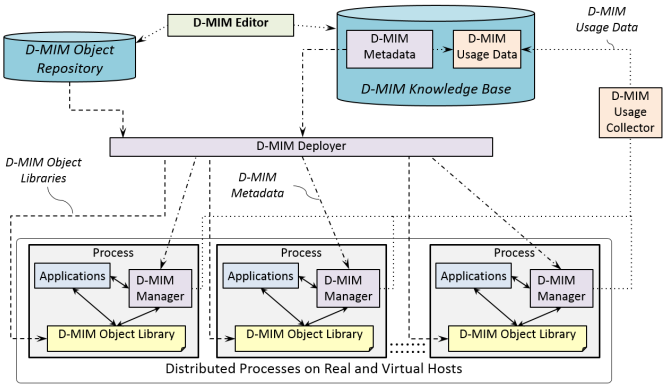

Fig. 1: Distributed Management Information Model (D-MIM) Management

A number of interesting technologies have emerged in the computer science domain that can be used to share information between distributed processes. Memory Driven Computing [7] has been proposed as a way of using the large banks of cheap memory on the hosts of modern distributed systems. Implementations of Distributed Hash Tables [8] that exchange unstructured maps of information between processes have been available for some time [9][10]. In parallel, frameworks have emerged to support distributed synchronized locks across processes. Some frameworks take a centralized approach, where a central server holds a record of all locks and controls lock access [11] while others take a fully distributed approach, with lock data being exchanged between processes at run time [10]. In addition, distributed monitoring technologies such as syslog [12] and Log4j [13] and persistence such as relational databases and distributed file systems are mature and are very suitable for distributed model monitoring.

This paper describes our approach for Distributed Management Information Model (D-MIM) Management (Fig. 1). At design time, a D-MIM author designs D-MIMs using an editor which stores metadata describing the D-MIMs in a Model Knowledge Base and creates a D-MIM Object definition for

each concept in each D-MIM, which may be instantiated at run time. At run time, the D-MIM Deployer distributes D-MIM Metadata and D-MIM Objects to distributed processes on real or virtual hosts. The Deployer deploys only the required set of D-MIMs to a given process, ensures that the D-MIMs on each host are consistent, upgrades D-MIMs on processes as the D-MIM definitions change in the knowledge base, and removes D-MIMs when they are no longer required for a process. D-MIMs can then be used to share information within an application instance, across instances of the same or different applications in a single process, or across application instances in multiple (possibly distributed) processes.

The D-MIM Manager for a process uses D-MIM Metadata to build local copies of only those D-MIMs required by the applications running in that process. It controls reads and writes to D-MIMs by applications in the process and uses a distribution mechanism (e.g. Inifinspan[9] or Hazelcast[10]) to transfer updates to the other processes using that same D-MIM. It also controls D-MIM access by applications; which applications have read or write access to different parts of a D-MIM is defined in the D-MIM metadata. If an application requires synchronized read or write access to a D-MIM concept instance, the D-MIM Manager uses a locking mechanism (e.g. Apache Curator [11] or Hazelcast Locking [10]) to ensure that the instance is appropriately locked or unlocked over all processes using that D-MIM concept instance. Each D-MIM Manager logs initializations, reads, writes, locks, and unlocks on D-MIM concept instances and sends those logs to a D-MIM Usage Collector for storage in the D-MIM Knowledge base. These logs audit the usage of every D-MIM concept instance in every D-MIM on every process, allowing the consistency of each D-MIM concept instance to be verified. The D-MIM metadata also enables consistent persistence of information in D-MIMs; it describes what D-MIM concept instances are persisted, the conditions when they are persisted, which processes should persist data, and the type of persistent store that should be used.

In this paper we survey related work in §II and in §III, we describe our approach for Management Information Model distribution. §IV shows a usage scenario of our approach and §V describes our evaluation of the approach.

II. Background and Related Work

There is no existing consistent method for defining, scoping, and using a distributed management information model with a scope wider than a single NE. Although hierarchies of models are very common, each model is a separate standalone instance with responsibility for managing the information of the NE in which the information model instance resides. The problem of information model coordination is magnified because a different approach to model coordination is taken in each domain. For example, a given radio network cell C1 is modeled in a base station model as cell `C1` and is again modeled separately in the management system as cell `C1`. The models are different, the cell is modeled twice, and the management system and base station NEs must interact with each other to keep those models consistent, in this case using 3GPP IRPs [4]. Likewise, a network interface is modeled in a computer using MIB-II [5] as `eth0` and again separately in an element manager as `eth0`. Again, the element manager and host computer must interact with each other to keep their models consistent, in this case using SNMP.

Information model coordination has been a challenge in management systems for many years, exasperated by the advent of Soft Networking. All current solutions rely on a stable hierarchy being present, with NEs occasionally entering and leaving the hierarchy in a controlled manner. When resources are managed across transient (and often virtual) NEs, where NEs, element managers and even network managers are transient, spun up and torn down as required, a hierarchical approach to model coordination makes it very difficult to maintain a consistent distributed view of those models. The approach of having independent static hierarchical models envisaged for telecommunication networks of the 1980s [6] does not operate well in today's dynamic networks due to the overhead and latency introduced by model coordination.

Although Distributed Shared Memory [14], Memory Driven Computing [7] and distributed cache frameworks [9][10] allow memory to be shared between systems, the data that is shared is unstructured and uncontrolled, usually as distributed maps. These are too unstructured and uncoordinated to be used safely by management applications. Each process has the freedom to add, modify, or remove entire distributed maps or the instances in the maps. For example, process `P1` adds an instance named `C1` as an object of type `cell` with parameters representing various cell attributes. Process `P2` can simply overwrite the instance `C1` with a `string` value or even delete the instance, or worse, a process can place a completely incompatible `customer` object as a value for `C1` in the map.

Data sharing using databases is common, but a centralized database for multiple distributed applications can introduce significant latency, even with caching mechanisms like memcached (https://memcached.org/). There are numerous approaches to use distributed databases for distributed data access, e.g. CouchBase (http://www.couchbase.com) is a distributed NoSQL database, but like most other NoSQL databases data models schemas cannot be enforced. Many traditional relational databases support replication using either master-master or master-slave replication strategies, but most such database replication strategies are only loosely consistent, i.e. asynchronous with lazy or eventual consistency, violating ACID properties, or where eager replication with enforced consistency is used then updates introduce high latency.

The ability to lock a particular element in a model to allow a management application preform a safe read or write is a fundamental requirement for any distributed consistent data approach. Distributed locking mechanisms [11][10] support synchronized locking of named distributed locks, but they do not provide a mechanism to associate or bind a named lock to a particular model concept instance, so controlled distributed locking of named model concept instances is not supported. A straightforward locking approach is preferred to a transaction

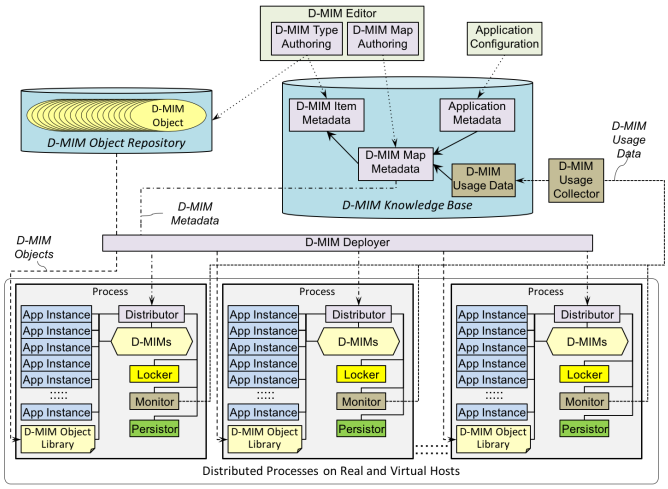

Fig. 2: Distributed Management Information Model (MIM) Architecture

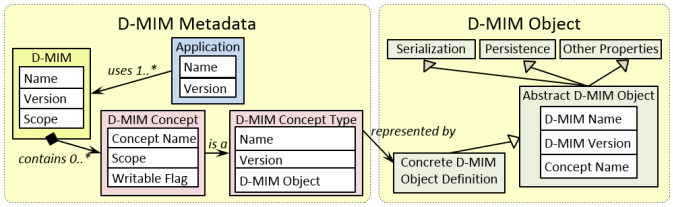

Fig. 3: Metadata and Objects for Distributed MIM Management

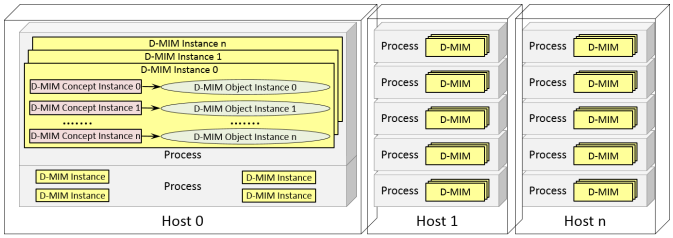

Fig. 4: D-MIMs Deployed to Processes on Hosts

approach because transaction frameworks such as [15] and [16] introduce a high degree of complexity and coordination that cannot be hidden from management applications. Explicit support for starting, joining, committing, and aborting transactions must be provided. Further, transaction approaches do not scale well, as the number of application instances increase the speed of transaction execution diminishes rapidly [17][18].

Common monitoring of operations on shared models is difficult where no common model exists. For example, each change on `cell C1` or `interface eth0` must be independently and separately logged by both the manager and the NE using their representation of `C1` and `eth0`. These separate logs must then be aggregated and correlated to provide a complete view of model concept instance initializations, writes, reads, and deletions. Similarly, if used, model persistence must also be done separately. Each separate model is independently persisted and managed, where the separate models must then be mapped and joined to provide a common view.

III. Distributed Management Information Models

Fig. 2 shows the architecture of Distributed Management Information Model Management. At design time, D-MIMs are defined in metadata and stored in the D-MIM Knowledge base using an editor. The D-MIM Deployer distributes the D-MIMs to a D-MIM Manager in each process that is using Distributed MIM management. Each process may host one or more instances of one or more applications. The D-MIM Manager manages its local copies of D-MIMs, provides access to the local MIMs for applications, distributes the MIM contents to other processes using a distribution mechanism (e.g. [9] or [10]), as well as managing locking, monitoring, and persisting of MIM Model concept Instances.

A. Distributed Management Information Model Structure

The Metadata and Object structure for D-MIMs is shown in Fig. 3. A D-MIM is composed of D-MIM concepts, each of which has a Concept Type. The Concept Type is represented by a D-MIM Object, which is instantiated at run time and holds the MIM concept instance value. For example, the D-MIM for a certain type of base station has D-MIM concepts instances of type `OwnedCell` for each of the cells it owns and have D-MIM concept instances of type `NeighbourCell` for each of the cells adjacent to the base station's cells. The concept `OwnedCell` and `NeighbourCell` are both represented by the `Cell` D-MIM Concept type, and their value will be held in a `Cell` D-MIM object at runtime.

The name of each D-MIM, D-MIM Concept Type, and Application is unique in a D-MIM deployment and is used for referencing their appropriate metadata. Scope defines application visibility: `Application` scope gives visibility only to explicitly defined applications, `Global` scope give read and modify to any application, and `External` scope is read-only, but these may be modified by external systems. Other more selective scopes can also be defined.

A serializable D-MIM Object implements a D-MIM concept and holds its concrete value. D-MIM Objects can therefore be transferred between processes and be persisted. D-MIM Object definitions contain any property definitions that is required to implement its domain model, for example a `Cell` D-MIM Object may have a `Cell_Id` property, and a `NetworkInterface` D-MIM Object may have an `ifPhysAddress` property.

Application Metadata (see Fig. 2) specifies what applications use D-MIM management and list the D-MIMs used by each application. Therefore, given a list of applications to be run in a process, the D-MIM deployer and D-MIM manager can infer which D-MIMs should be available to that process.

B. D-MIM Deployment

Fig. 4 shows an exploded view of three D-MIM instances deployed on `Process_0` on `Host_0`. A D-MIM instance copy is held as a map in a distributed hash table that is shared across all processes that use that D-MIM. Identical D-MIM instances to those in Fig. 4 exist in all other processes on all hosts that use those D-MIMs. Each concept instance in a D-MIM is instantiated as a map entry whose value is a D-MIM object instance that holds the value of the D-MIM concept.

The Distributed Model Deployer shown in Fig. 2 distributes D-MIM metadata and Objects to processes. When a process starts, the D-MIM Deployer requests a list of the applications on the process, sends the metadata and D-MIM objects for all D-MIMs used by those applications to the D-MIM Manager

of the process, and asks the D-MIM Manager on the process to start Distributed MIM Management. If, during execution, a new application is installed in a process or an application is removed from a process, the D-MIM Deployer adds or removes D-MIM metadata and D-MIM objects as appropriate and requests the D-MIM Manager for the process to perform an update. The D-MIM Deployer also ensures consistency of D-MIM metadata and Object Libraries across processes. If D-MIM metadata is updated in the Model Knowledge Base, the Deployer coordinates propagation of that update across the D-MIM Managers in all processes that use the updated D-MIM.

1) D-MIM Creation in a Process: To create a D-MIM the distributor in the D-MIM Manager stores the D-MIM objects to its Object Library. It then checks if the distribution mechanism already has a map for that D-MIM (i.e. another process may have already defined the D-MIM distributed map). If a D-MIM map is already available, the distributor simply reads that map. Otherwise the distributor creates a new map, then iterates over each D-MIM concept, using the D-MIM object definition for the concept from the D-MIM Object Library to create a D-MIM object instance for the concept in the distributed map. The distributor then checks if a current value for the D-MIM Object instance is available from persistent store, otherwise, a default value is used.

2) D-MIM Deletion on a Process: To delete a D-MIM the distributor in the D-MIM Manager iterates over each concept in the D-MIM and deletes the D-MIM Object Instance from the D-MIM distributed map in the distribution mechanism. If persistence was active, it first saves the value of the D-MIM object instance. When all instance are deleted, the distributor deletes the D-MIM in the distribution mechanism. Finally, the D-MIM Objects are removed from the Object Library.

3) D-MIM Update Propagation: The deployer requests the D-MIM Manager on each process using the D-MIM to lock its copy, thus blocking all access to all D-MIM copies. Versions updates are compatible if the new version only specifies additional concepts to D-MIMs and/or only additions or extensions of D-MIM Objects. If an update is compatible, the Deployer requests the D-MIM manager in the process to perform a D-MIM update; otherwise it requests the D-MIM manager to perform a D-MIM deletion followed by a D-MIM creation. Once all processes have updated or recreated their D-MIMs, the Deployer request all relevant D-MIM managers to release its D-MIM lock.

4) D-MIM Update for a Process: The distributor in the D-MIM Manager updates the Object Library with any new D-MIM objects. It then checks if the version of the D-MIM in the distribution mechanism has changed. If so, the distributor reads it from the distribution mechanism. Otherwise, the distributor iterates over each new or updated D-MIM concept, using the concept's D-MIM object definition from the D-MIM Object Library to create or replace the concept's D-MIM object instance in the distributed map. The D-MIM object instance value is set to the value of the replaced object instance, to a value from persistent storage, or to a default value.

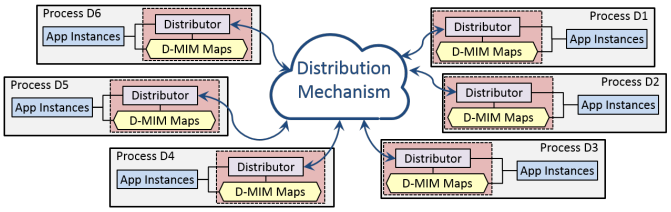

Fig. 5: Distribution of Model Maps for MIMs across Processes

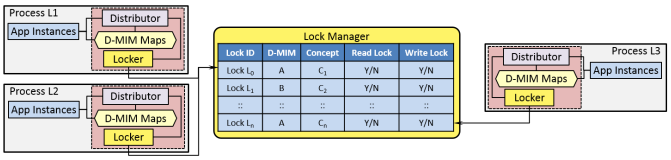

Fig. 6: Locking of D-MIM Concepts across Processes

C. D-MIMs at Run Time

The distributor in each D-MIM Manager (Fig. 5) manages the D-MIMs for the applications in its process using distributed hash maps which are shared with other processes using the underlying distribution mechanism. The (unstructured) distribution mechanism can be used because directly the distributor controls both the structure of and access to distributed D-MIM Maps using the D-MIM metadata. Applications can read or write D-MIMs using a key-value based `Map` API provided by the distributor. For example, when an application instance changes object instance `O` on D-MIM map `M` on process `D6`, the distributor on process `D6` propagates that change to the distribution mechanism, which updates object instance `O` on D-MIM map `M` in processes `D1`–`D5` of the Distributed MIM Management system.

1) D-MIM Locking: Any distributed MIM must have integrated support for synchronized distributed locking for safe and controlled read and write access to MIM concepts. A simple approach could use a global exclusive lock. However, such an unacceptable approach would cause all accesses to be sequential. Instead a fine-grained lock for each D-MIM concept is provided.

Fig. 6 shows how an underlying distributed locking mechanism such as Curator [11] or Hazelcast Locking [10] is used to associate a distinct distributed lock with each distributed D-MIM concept across processes. Because the Lock Manager of the underlying locking mechanism already provides distributed locks, only one application in a process can hold a lock at any one time. Application `A` on Process `P1` can acquire lock `L0` on D-MIM `A` concept `C1` without affecting access to any other concepts on D-MIM `A` (e.g. application `C` on process `P3` can separately acquire lock `Ln` on concept `Cn` on D-MIM `A`). However, if another application requests a lock on D-MIM `A` concept `C1` its request is queued while application `A` holds the lock. Read and write locks are supported, where read locks can be shared but a write lock is exclusive.

2) Monitoring Operations on D-MIM Concept instances: As shown in Fig. 7, D-MIM Management uses an underlying monitoring mechanism such as Log4j [13] or syslog [12] to log operations. The distributor logs D-MIM concept instance initializations, deletions, reads, writes and read and write lock acquisitions and releases. The D-MIM Usage Collector receives these logs from the underlying mechanism and stores

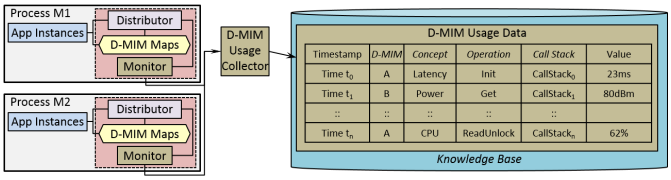

Fig. 7: Monitoring Operations on D-MIM concept instances

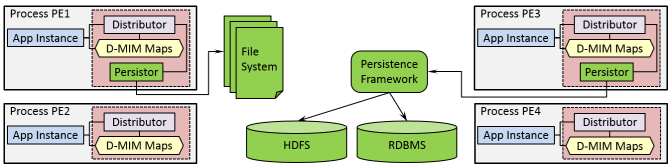

Fig. 8: Persisting Model Maps for MIMs in Processes

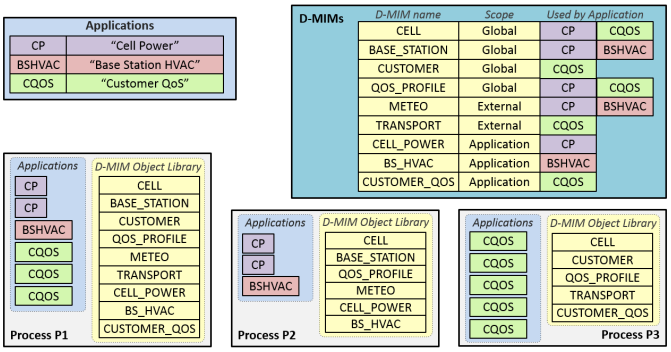

Fig. 9: Examples of Distributed MIM Management Models

them in the Knowledge Base in the format shown in Fig. 7. Each log entry contains the D-MIM concept instance identity, a timestamp, the operation performed, the application call-stack, and the pre- and post-value of the instance. Applications can also add further context to the call-stack to improve logging.

Because the monitored information gives the usage of every D-MIM concept instance by every application on every host, the consistency of each D-MIM individual can be verified. Consider the case where the `Power` concept instance of cell `C1` may be changed by separately by two separate applications, e.g. a coverage optimization application attempts to *increase* `Power` and an energy saving application attempts to *decrease* `Power`. Interference between the two applications (based on the `Power` concept instance) can be easily detected and audited. In addition, because the usage information of all D-MIM concept instances is available, analytics can be applied to the usage information to identify less obvious access patterns, conflicts and side effects.

3) Persisting D-MIMs: Persistence may be activated on a process using D-MIM. In Fig. 8, persistence is active on processes `PE1` and `PE3` but not `PE2` and `PE4`. D-MIM Management can use the file system for persistence (process `PE1`) or use an underlying persistence mechanism such as a Database or distributed storage system (process `PE3`) to store D-MIM information. It is not necessary to activate persistence on all processes because it is adequate to save a one copy of a D-MIM to the persistent store (or more for redundancy). When persistence is activated on a process, the distributor calls the persistor periodically to persist the D-MIMs.

TABLE I: Distribution Mechanisms

Mechanism	Comment
Memory	Java Hash Maps, only for use in one JVM (Default)
Infinispan	Inifinispan distributed maps [9]
Hazelcast	Hazelcast distributed maps [10]

TABLE II: Locking Mechanisms

Mechanism	Comment
Java	Java Locks, only for locking in one JVM (Default)
Curator	Apache Curator locks [11] as used in Zookeeper
Hazelcast	Hazelcast distributed locks [10]

IV. D-MIM Usage Scenario

Fig. 9 illustrates a usage scenario for D-MIMs. Multiple instances of three *applications*, `CP` (Cell Power), `BSHVAC` (Base Station Heating, Ventilation, and Air Conditioning), and `CQOS` (Customer QoS), are running on distributed *processes* `P1`, `P2`, and `P3`, with 9 D-MIMs in the system (Fig. 9). The `CELL` D-MIM holds information on cells, with *Global* scope, and is used by the `CP` and `CQOS` applications. The `TRANSPORT` D-MIM contains timetable and status information for motorways, railways, and airlines, with *External* scope. The `BASE_STATION` D-MIM holds management data about the base station hardware. Each application also has its own application D-MIM, (e.g. `BSHVAC` D-MIM for the `BSHVAC` application), which holds internal state information for use by that application only (*Application* scope). `BSHVAC` application D-MIM instances may hold information that is specific to just that application such as the current fan speeds (RPM) in each base station.

Instances of all 3 applications are running in process `P1` so the Distributed Model Deployer (see Fig. 2) sends the metadata for all D-MIMs to that process and the D-MIM Manager for `P1` instantiates distributed maps for each D-MIM. Process `P2` runs 2 instances of the `CP` application and and 1 instance of the `BSHVAC` applications, therefore 6 required D-MIMs are deployed and initialized. Process `P3` is runs 5 instances of application `CQOS` only, so only the 5 D-MIMs used by the `CQOS` application are deployed and used. At runtime, each application instance uses the D-MIM distributed maps as if they were local, and is not aware that the maps are distributed, with D-MIM Management ensuring that all locks and updates are distributed correctly. If any application instance writes to or locks any shared D-MIM map, it is consistently propagated and made available to all other applications using that instance.

V. Evaluation

We evaluated our implementation to assess the feasibility and performance of distribution and locking of D-MIMs. We used the configurable distribution and locking mechanisms listed in Tables I and II respectively. The JVM-native *Memory* and *Java* distribution and locking mechanisms work with processes in a single JVM only and are included for evaluation and testing purposes, while the other distribution and locking mechanisms support distributed processes.

In each run, we created a D-MIM of integers. We selected distribution and locking mechanism pairs and assessed how well each mechanism pair worked in reading the D-MIM with and without read locks (since dirty reads are perfectly acceptable in some applications), and writing the D-MIM with

TABLE III: Evaluation Run Combinations

Combination	Values
D-MIM Sizes	1 to 65536
32 Threads	1JVM/32Thread, 2JVM/16Thread, 4JVM/8Thread
D-MIM Usage	1000 Random Reads or Writes per Thread
Dist/Lock	Haz/Cur, Haz/Haz, Inf/Cur, Inf/Haz, Mem/java, Mem/Cur, Mem/Haz

write locks. To measure how each pair performed as D-MIM size increased, we evaluated them with D-MIM sizes varying by powers of 2 from 2^0 to 2^{16}. 32 Applications (threads) were executed in each run; executing in a single JVM (32 thread in the JVM), in two JVMs (16 threads per JVM), or in four JVMs (8 threads per JVM). Each mechanism pair was evaluated with each D-MIM size in all thread configurations. The possible evaluation run combinations are shown in Table III.

Each thread performing 1000 reads (and writes for write tests) on random integer entries in the D-MIM. For read lock tests, threads read the value of the selected D-MIM integer. For write lock tests, threads read and incremented the value of the selected D-MIM integer. We executed our evaluation on a MacBook Pro laptop with at 2.6GHz 8-Core i7 CPU with 16GB of memory running MacOS version 10.12.

We ran 749 tests in our evaluation. No concurrency and locking errors were observed during test execution or in test logs. We verified that the sum of the integers in the D-MIM was 32,000 at the end of each test run. This demonstrated that the D-MIM approach, and its distribution and locking mechanisms are stable, consistent, safe, and correct.

The plots on Fig. 10, 11 and 12 show execution times for non locking and locking reads and locking writes for distribution/locking mechanism pairs for the three JVM/thread combinations. In-memory distribution and Java locking is not included in Fig. 11 and 12 because those mechanisms cannot be used where applications are deployed in multiple JVMs.

In Fig. 10a, one can see that, except for hazelcast distribution, all mechanisms perform vey well, as one would expect for in-memory non-locked reads. The performance of Hazelcast deteriorates rapidly as D-MIM size increases. Fig. 10b and 10c demonstrate how performance deteriorates for the fully fledged mechanisms once locking is introduced. As one would expect, in-memory distribution and locking is significantly more performant. It is interesting to note that the performance of read and write locks are similar for all combinations. Performance of all combinations deteriorates as D-MIM size increases. The Inifinispan/Hazelcast pair has much better performance than other fully fledged mechanisms, indeed the performance of this combination matches the Memory/Java combination.

Fig. 11a and Fig. 12a show that distribution performs very well when locking is not active, even over multiple JVMs. However, Hazelcast performance deterioration as D-MIM size increases is evident. Introducing read locking (Fig. 11b and Fig. 12b) causes a deterioration in performance, as one would expect. The overhead for map and lock distribution increases as D-MIM size increases. The combination of Infinispan distribution and Hazelcast locking is the most robust to D-MIM size increase, probably because Hazelcast distributed locking is more scalable than Apache Curator centralised locking, where every lock operation interacts with the Zookeeper server.

The plots for write lock performance in multi-JVM cases (Fig. 11c and Fig. 12c) show a `U` shaped pattern. At very low D-MIM sizes, high contention for access from 32 threads to 1, 2, 4, or 8, D-MIM integer instances forces threads to wait while other threads hold D-MIM locks. As D-MIM size increases, contention between processes decreases. When the number of processes equals the number of D-MIM instances, the curves level off. As with read locking, as D-MIM size increases, distribution and locking overhead causes a corresponding decrease in performance. As with read locking, the Infinspan/Hazelcast pair demonstrates the best performance.

The D-MIM approach and its distribution/locking implementation was stable, error free and correct for all of its 749 test runs. The performance measurements shown in Fig. 10, 11 and 12 were taken on a laptop computer, and no special efforts were taken to tune or optimise the distribution or locking mechanisms. Even with these caveats, and given that the evaluation used an extreme scenario where lock contention was very high, the performance of all the distribution/locking combinations in our implementation is very promising.

VI. Conclusions and Future Work

In this paper we have presented Distributed Management Information Models (D-MIMs). D-MIMs manage the distribution of formally modeled management information in a way that allows distributed management applications to use the D-MIM information in a safe, controlled, and audited manner. D-MIMs provide a unified view of modeled information across distributed applications and processes. Access to D-MIMs is controlled; D-MIMs are distributed only to those processes and applications that are specified as using them. The D-MIM distributor in each process enforces controlled reading, writing, locking, and monitoring of information in the distributed management information models. Consistency is inherent in D-MIMs because applications work towards common distributed MIMs. In addition, each operation on a D-MIM concept instance is logged by every process that uses the D-MIM concept instance to allow the consistency of usage to be verified and interference to be identified. Our evaluation demonstrated the stability and performance of the D-MIM approach when deployed using a number of distribution and locking mechanisms.

Acknowledgement

This work is partly funded by the European Commission via the ARCFIRE project (Grant 687871) under the H2020 program.

References

[1] ETSI, "Network Functions Virtualisation (NFV); Management and Orchestration," ETSI, Tech. Rep. GS NFV-MAN 001, December 2014.

[2] Ericsson. (2016, Feb) Ericsson network manager (enm). Ericsson.

[3] ITU-T, "Generic network information model," ITU-T, Tech. Rep. M.3100, Apr 2005.

[4] 3GPP, "Bulk CM Integration Reference Point (IRP): Information Service (IS)," TS 32.612, 3GPP, Tech. Rep. 3GPP TS 32:612, January 2011.

[5] IETF, "Management information base for network management of tcp/ip-based internets: Mib-ii," IETF, Tech. Rep. RFC 1213, March 1991.

[6] ITU-T, "Principles for a telecommunications management network," ITU-T, Tech. Rep. M.3010, Feb. 2000.

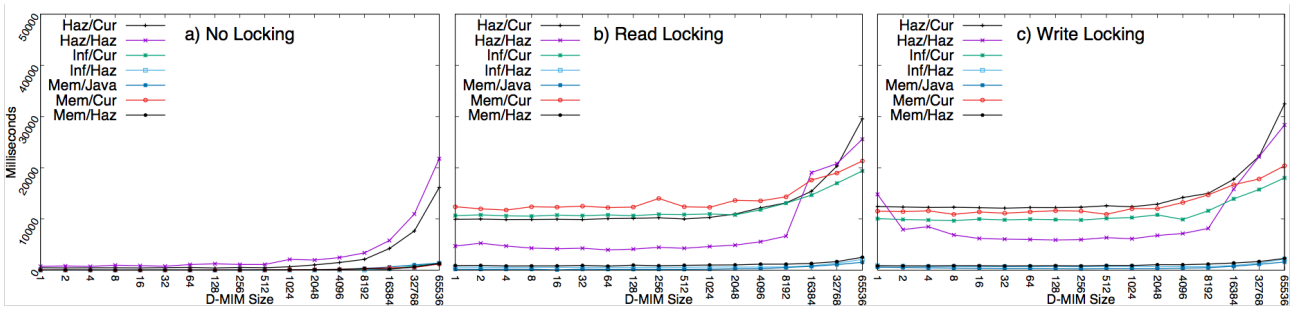

Fig. 10: Performance using 32 Threads in a single JVM (32 Threads per JVM)

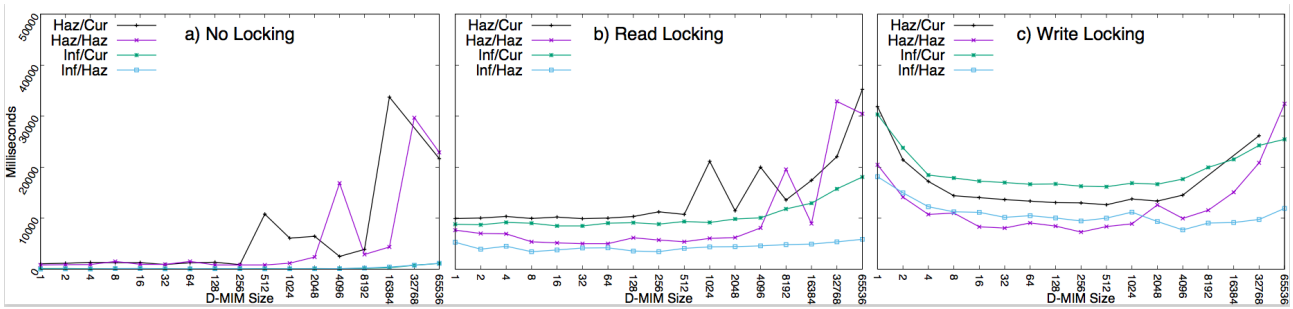

Fig. 11: Performance using 32 Threads over two JVMs (16 Threads per JVM)

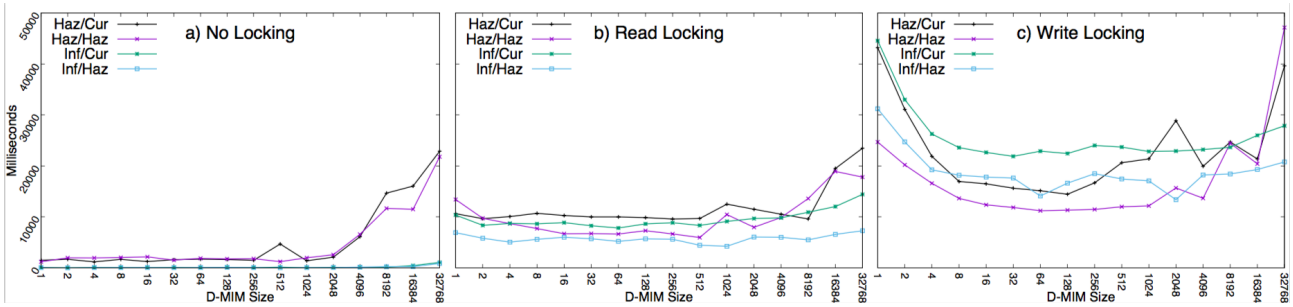

Fig. 12: Performance using 32 Threads over four JVMs (8 Threads per JVM)

[7] K. Bresniker, S. Singhal, and R. Williams, "Adapting to thrive in a new economy of memory abundance," *Computer*, vol. 48, no. 12, pp. 44–53, Dec 2015.

[8] G. Coulouris *et al.*, *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley, 2012.

[9] Infinispan, *Infinispan Documentation*, 8th ed., Infinispan, Feb 2016.

[10] Hazelcast, *Hazelcast Documentation*, 3rd ed., Hazelcast, February 2016.

[11] A. Curator, *Apache Curator Getting Started Guide*, 3rd ed., Apache Curator, Oct 2015.

[12] IETF, "The syslog protocol," IETF, Tech. Rep. RFC 5424, March 2009.

[13] A. Log4j, *Apache Log4j User's Guide*, 2nd ed., Apache Log4j, Dec 2015.

[14] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed shared memory: Concepts and systems," *Parallel Distributed Technology: Systems Applications, IEEE*, vol. 4, no. 2, pp. 63–71, Summer 1996.

[15] Oracle, *Java Transaction API (JTA)*, 1st ed., Oracle, April 1999.

[16] M. Little *et al.*, *Narayana Transaction Manager Project Documentation*, 2nd ed., jboss.org, January 2016.

[17] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and scalability of ejb applications," *SIGPLAN Not.*, vol. 37, no. 11, pp. 246–261, Nov. 2002.

[18] M. Femminella, E. Maccherani, and G. Reali, "Performance Management of Java-based SIP Application Servers," in *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, May 2011, pp. 493–500.