Decentralized Model Persistence for Distributed Computing

Abel Gómez, Amine Benelallam, Massimo Tisi

Decentralized Model Persistence for Distributed Computing

Abel Gómez, Amine Benelallam, and Massimo Tisi

AtlanMod team (Inria, Mines Nantes, LINA), France
{abel.gomez-llana|amine.benelallam|massimo.tisi}@inria.fr

Abstract. The necessity of manipulating very large amounts of data and the wide availability of computational resources on the Cloud is boosting the popularity of distributed computing in industry. The applicability of model-driven engineering in such scenarios is hampered today by the lack of an efficient model-persistence framework for distributed computing. In this paper we present NeoEMF/HBase, a persistence backend for the Eclipse Modeling Framework (EMF) built on top of the Apache HBase data store. Model distribution is hidden from client applications, that are transparently provided with the model elements they navigate. Access to remote model elements is decentralized, avoiding the bottleneck of a single access point. The persistence model is based on key-value stores that allow for efficient on-demand model persistence.

Keywords: Model Persistence, Key-Value Stores, Distributed Persistence, Distributed Computing

1 Introduction

The availability of large data processing and storage in the Cloud is becoming a key resource for part of today's industry, within and outside IT. It offers a tempting alternative for companies to process, analyze, and discover new data insights, yet in a cost-efficient manner. Thanks to existing Cloud computing companies, this facility is extensively available for rent [11]. This ready-to-use IT infrastructure is equipped with a wide range of distributed processing frameworks, for companies that have to occasionally process large amounts of data.

One of the principal ingredients behind the success of distributed processing are distributed storage systems. They are designed to answer to data processing requirements of distributed and computationally extensive applications, i.e., wide applicability, scalability, and high performance. Appearing along with MapReduce [10], BigTable [9] strongly stood in for these qualifications. One of the most compliant open-source implementations of MapReduce and BigTable are Apache's Hadoop [17] and HBase [18], respectively.

Another success factor for widespread distributed processing is the appearance of high-level languages for simplifying distribution by a user-friendly syntax (mostly SQL-like). They transparently convert high-level queries into a series of

parallelizable jobs that can run in distributed frameworks, such as MapReduce, therefore making distributed application development convenient.

We believe that Model-Driven Engineering (MDE), especially the query/-transformation languages and engines, would be suitable for developing distributed applications on top of structured data (models). Unfortunately, MDE misses some fundamental bricks towards building fully distributed transformation/query engines. In this paper we address one of those components, i.e. a model-persistence framework for distributed computing. Several distributed model-persistence frameworks exist today [3,16]: for the Eclipse Modeling Framework (EMF) [6] two examples are Connected Data Objects (CDO) [3] that is based on object relational mapping[1], and *EMF fragments* [15], that maps large chunks of model to separate URIs. We argue that these solutions are not well-suited for distributed computing, exhibiting one or more of the following faults:

- Model distribution is not transparent: so queries and transformations need to explicitly take into account that they are running on a part of the model and not the whole model (e.g. *EMF fragments*)
- Even when model elements are stored in different nodes, access to model elements is centralized, since elements are requested from and provided by a central server (e.g. CDO over a distributed database). This constitutes a bottleneck and does not exploit a possible alignment between data distribution and computation distribution.
- The persistence backend is not optimized for atomic operations of model handling APIs. In particular files (e.g. XMI over HDFS [7]), relational databases or graph databases are widely used while we have shown in previous work [12] that key-value stores are very efficient in typical queries over very large models. Moreover key-value stores are more easily distributed with respect to other formats, such as graphs.
- The backend assumes to split the model in balanced chunks (e.g. *EMF Fragments*). This may not be suited to distributed processing, where the optimization of computation distribution may require uneven data distribution.

In this paper we present NeoEMF/HBase, a persistence backend for EMF built on top of the Apache HBase data store. NeoEMF/HBase is transparent w.r.t. model manipulation operations, decentralized, and based on key-value stores. The tool is open-source and publicly available at the paper's website[2]. This paper is organized as follows: Section 2 presents HBase concepts and architecture, Section 3 presents the NeoEMF/HBase architecture, data model and properties; and finally, Section 4 concludes the paper and outlines future work.

[1] CDO servers (usually called *repositories*) are built on top of different data storage solutions (ranging from relational databases to document-oriented databases). However, in practice, only relational databases are commonly used, and indeed, only *DB Store* [1], which uses a proprietary Object/Relational mapper, supports all the features of CDO and is regularly released in the *Eclipse Simultaneous Release* [2,4,5].

[2] `http://www.emn.fr/z-info/atlanmod/index.php/NeoEMF/HBase`

2 Background: Apache HBase

Apache HBase [18] is the Hadoop [17] database, a distributed, scalable, versioned and non-relational big data store. It can be considered an open-source implementation of Google's Bigtable proposal [9].

2.1 HBase data model

In HBase, data is stored in tables, which are sparse, distributed, persistent multidimensional sorted maps. A map is indexed by a row key, acolumn key, and a timestamp. Each value in the map is an uninterpreted array of bytes.

HBase is built on top of the following concepts [14]:

Table — Tables have a name, and are the top-level organization unit for data in HBase.

Row — Within a table, data is stored in *rows*. Rows are uniquely identified by their *row key*.

Column Family — Data within a row is grouped by *column family*. Column families are defined at table creation and are not easily modified. Every row in a table has the same column families, although a row does not need to store data in all its families.

Column Qualifier — Data within a column family is addressed via its *column qualifier*. Column qualifiers do not need to be specified in advance and do not need to be consistent between rows.

Cell — A combination of *row key*, *column family*, and *column qualifier* uniquely identifies a *cell*. The data stored in a cell is referred to as that cell's value. Values do not have a data type and are always treated as a `byte[]`.

Timestamp — Values within a cell are versioned. Versions are identified by their version number, which by default is the timestamp of when the cell was written. If the timestamp is not specified for a read, the latest one is returned. The number of cell value versions retained by HBase is configured for each column family (the default number of cell versions is three).

Figure 1 (extracted from [9]) shows an excerpt of an example table that stores Web pages. The row name is a reversed URL. The `contents` column family contains the page contents, and the `anchor` column family contains the

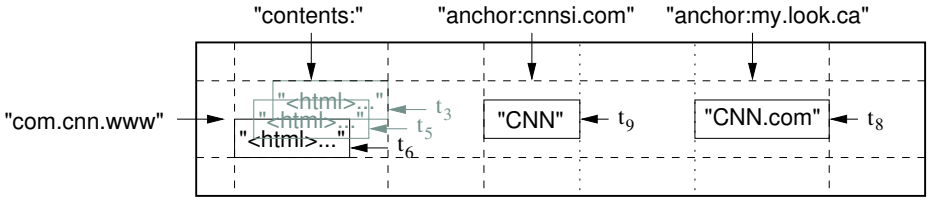

Fig. 1: Example of a table in HBase/BigTable (extracted from [9])

text of any anchors that reference the page. CNN's home page is referenced by both the *Sports Illustrated* and the *MY-look* home pages, so the row contains columns named `anchor:cnnsi.com` and `anchor:my.look.ca`. Each anchor cell has one version; the `contents` column has three versions, at timestamps t_3, t_5, and t_6.

2.2 HBase architecture

Fig. 2 shows how HBase is combined with other Apache technologies to store and lookup data. Whilst HBase leans on HDFS to store different kind of configurable size files, ZooKeeper [19] is used for coordination. Two kinds of nodes can be found in an HBase setup, the so-called *HMaster* and the *HRegionServer*. The *HMaster* is the responsible for assigning the regions (*HRegions*) to each *HRegionServer* when HBase is starting. Each *HRegion* stores a set of rows separated in multiple column families, and each column family is hosted in an *HStore*. In HBase, row modifications are tracked by two different kinds of resources, the *HLog* and the *Stores*. The *HLog* is a store for the write-ahead log (WAL), and is persisted into the distributed file system. The WAL records all changes to data in HBase, and in the case of a *HRegionServer* crash ensures that the changes to

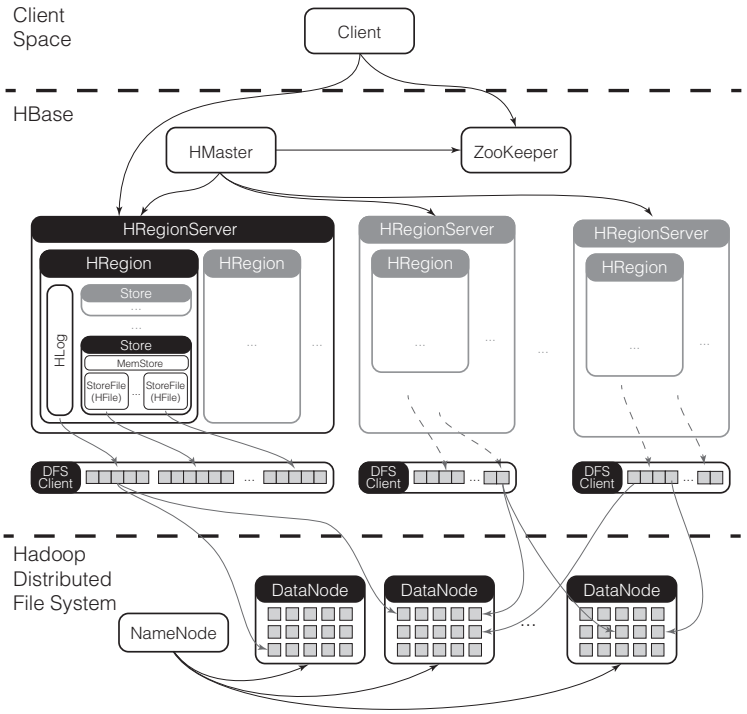

Fig. 2: HBase architecture

the data can be replayed. Stores in a region contain an in-memory data store (*MemStore*) and a persistent data stores (*HFiles*, that are persisted into the distributed file system) *HFiles* are local to each region, and used for actual data storage. The ZooKeeper cluster is responsible of providing the client with the information about both the *HRegionServer* and the *HRegion* hosting the row the client is looking up for. This information is cached at the client side, so that a direct communication could be directly setup for the next times without querying the *HMaster*. When an *HRegionServer* receives a write request, it sends the request to a specific *HRegion*. Once the request is processed, data is first written into the *MemStore* and when certain threshold is met, the *MemStore* gets flushed into an *HFile*.

2.3 HBase vs. HDFS

HDFS is the primary distributed storage used by Hadoop applications as it is designed to optimize distributed processing of multi-structured data. It is well suited for distributed storage and distributed processing using commodity hardware. It is fault tolerant, scalable, and extremely simple to expand. HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency, which makes it neither suitable nor optimized for atomic model operations. HBase is, on the other hand, a better choice for low-latency access. Moreover, HDFS resources cannot be written concurrently by multiple writers without locking and this results in locking delays. Also writes are always made at the end of the file. Thus, writing in the middle of a file (e.g. changing a value of a feature) involves rewriting the whole file, leading to more significant delays. On the contrary, HBase allows fast random reads and writes. HBase is row-level atomic, i.e. inter-row operations are not atomic, which might lead to a dirty read depending on the data model used. Additionally, HBase only provides five basic data operations (namely, *Get*, *Put*, *Delete*, *Scan*, and *Increment*), meaning that complex operations are delegated to the client appliaction (which, in turn, must implement them as a combination of these simple operations).

3 NeoEMF/HBase

Figure 3 shows the high-level architecture of our proposal for the EMF framework. It consists in a transparent persistence manager behind the model-management interface, so that tools built over the modeling framework would be unaware of it. The persistence manager communicates with the underlying database by a driver, and supports a pluggable caching strategy. In particular we implement the NeoEMF/HBase tool as a persistence manager for EMF on top of HBase and ZooKeeper. NeoEMF also supports other technologies, such as an embedded graph backend [8] and an embedded key-value store [12].

This architecture guarantees that the solution integrates well with the modeling ecosystem, by strictly complying with the EMF API. Additionally, the

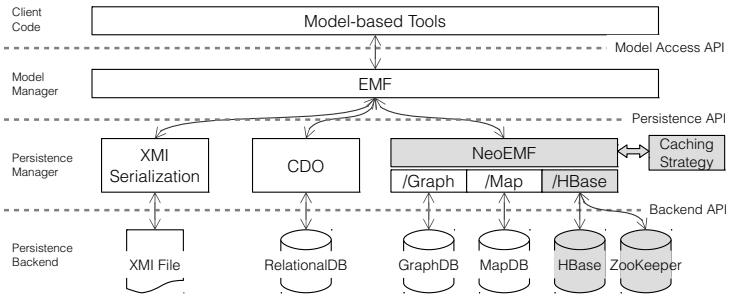

Fig. 3: Overview of the model-persistence framework

APIs are consistent between the model-management framework and the persistence driver, keeping the low-level data structures and code accessing the database engine completely decoupled from the modeling framework high level code. Maintaining these uniform APIs between the different levels allows including additional functionality on top of the persistence driver by using the decorator pattern, such as different cache levels.

NeoEMF/HBase offers lightweight on-demand loading and efficient garbage collection. Model changes are automatically reflected in the underlying storage, making changes visible to all the clients. To do so, first we decouple dependencies among objects by assigning a *unique identifier* to all model objects, and then:

- To implement *lightweight on-demand loading and saving*, for each live model object, we create a lightweight delegate object that is in charge of on-demand loading the element data and keeping track of the element's state. Delegates load and save data from the persistence backend by using the object's unique identifier.
- For *efficient garbage collection* in the *Java Runtime Environment*, we avoid to maintain hard Java references among model objects, so that the garbage collector can deallocate any model object that is not directly referenced by the application.

3.1 Map-based data model

We have designed the underlying data model of NeoEMF/HBase to minimize the data interactions of each method of the EMF model access API. The design takes advantage of the unique identifier defined in the previous section to flatten the graph structure into a set of key-value mappings.

Fig. 4a shows a small excerpt of a possible *Java* metamodel that we will use to exemplify the data model. This metamodel describes *Java* programs in terms of *Packages*, *ClassDeclarations*, *BodyDeclarations*, and *Modifiers*. A *Package* is a named container that groups a set of *ClassDeclarations* through the *ownedElements* composition. A *ClassDeclaration* contains a *name* and a set of

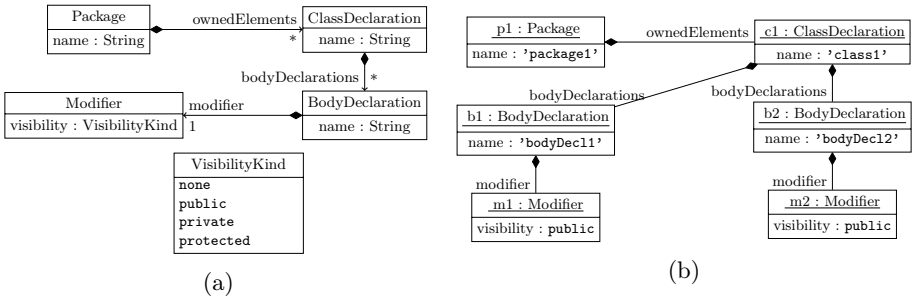

Fig. 4: Excerpt of the *Java* metamodel (4a) and sample instance (4b)

BodyDeclarations. Finally, a *BodyDeclaration* contains a *name*, and its *visibility* is described by a single *Modifier*.

Fig. 4b shows a sample instance of the *Java* metamodel, i.e., a graph of objects conforming with the metamodel structure. The model contains a single *Package* (`package1`), containing only one *ClassDeclaration* (`class1`). The *Class* contains the `bodyDecl1` and `bodyDecl2` *BodyDeclaration*s. Both of them are `public`.

NeoEMF/HBase uses a single table with three column families to store models' information: (i) a *property column family*, that keeps all objects' data stored together; (ii) a *type column family*, that tracks how objects interact with the meta-level (such as the *instance of* relationships); and (iii) a *containment column family*, that defines the models' structure in terms of containment references. Table 1[3] shows how the sample instance in Fig. 4b is represented using this structure.

As Table 1 shows, row keys are the object *unique identifier*. The `property` *column family* stores the objects' actual data. As it can be seen, not all rows have a value for a given column. How data is stored depends on the *property type* and *cardinality* (i.e., upper bound). For example, values for single-valued attributes (like the *name*, which stored in the `name` column) are directly saved as a single literal value; while values for many-valued attributes are saved as an array of single literal values (Fig. 4b does not contain an example of this). Values for single-valued references, such as the *modifier* containment reference from *BodyDeclaration* to *Modifier*, are stored as a single value (corresponding to the identifier of the referenced object). Examples of this are the cells for ⟨`b1`, `modifier`⟩ and ⟨`b2`, `modifier`⟩ which contain the values `'m1'` and `'m2'` respectively. Finally, multi-valued references are stored as an array containing the literal identifiers of the referenced objects. An example of this is the *bodyDeclarations* containment reference, from *ClassDeclaration* to *BodyDeclaration*, that for the case of the *c1* object is stored as { `'b1'`, `'b2'` } in the ⟨`c1`, `bodyDeclarations`⟩ cell.

Structurally, EMF models are trees (a characteristic inherited from its XML-based representation). That implies that every non-volatile *object* (except the

[3] Actual rows have been split for improved readability

Table 1: Example instance stored as a sparse table in HBase

	property					
Key	eContents	name	ownedElements	bodyDeclarations	modifier	visibility
'ROOT'	'p1'					
'p1'		'package1'	{ 'c1' }			
'c1'		'class1'		{ 'b1', 'b2' }		
'b1'		'bodyDecl1'			'm1'	
'b2'		'bodyDecl'			'm2'	
'm1'						'public'
'm2'						'public'

	containment		type	
Key	container	feature	nsURI	EClass
'ROOT'			'http://java'	'RootEObject'
'p1'	'ROOT'	'eContents'	'http://java'	'Package'
'c1'	'p1'	'ownedElements'	'http://java'	'ClassDeclaration'
'b1'	'c1'	'bodyDeclarations'	'http://java'	'BodyDeclaration'
'b2'	'c1'	'bodyDeclarations'	'http://java'	'BodyDeclaration'
'm1'	'b1'	'modifiers'	'http://java'	'Modifier'
'm2'	'b2'	'modifiers'	'http://java'	'Modifier'

root *object*) must be contained within another *object* (i.e., referenced from another *object* via a containment *reference*). The `containment` *column family* maintains a record of which is the container for every persisted object. The `container` column records the identifier of the container object, while the `feature` column records the name of the *property* that relates the container object with the child object (i.e., the object to which the row corresponds). Table 1 shows that, for example, the container of the *Package p1* is `ROOT` through the *eContents* property (i.e., it is a root object and is not contained by any other object). In the next row we find the entry that describes that the *Class c1* is contained in the *Package p1* through the *ownedElements property*.

The `type` *column family* groups the type information by means of the `nsURI` and `EClass` columns. For example, the table specifies the element *p1* is an instance of the *Package* class of the *Java* metamodel (that is identified by the `http://java` *nsURI*).

3.2 ACID properties

NeoEMF/HBase is designed as a simple persistence layer that maintains the same semantics as the standard EMF. Modifications in models stored using NeoEMF/HBase are directly propagated to the underlying storage, making changes visible to all possible readers immediatly. As in standard EMF, no transactional support is explicitly provided, and as such, ACID properties [13] (Atomicity, Consistency, Isolation, Durability) are only supported at the object level:

Atomicity — Modifications on object's properties are atomic. Modifications involving changes in more than one object (e.g. bi-directional references), are not atomic.

Consistency — Modifications on object's properties are always consistent using a compare-and-swap mechanism. In the case of modifications involving changes in more than one object, consistency is only guaranteed when the model is modified to grow monotonically (i.e., only new information is added, and no already existing data is deleted nor modified).

Isolation — Reads on a given object always succeeds and always give a view of the object's latest valid state.

Durability — Modifications on a given object are always reflected in the underlying storage, even in the case of a *Data Node* failure, thanks to the replication capabilities provided by HBase.

These properties allow the use of NeoEMF/HBase as the persistence backend for distributed and concurrent model transformations, since reads in the source model are consistent and always success; and the creation of the target model is a building process that creates a model that grows monotonically.

4 Conclusion and Future Work

In this paper we have outlined NeoEMF/HBase, an on-demand, memory-friendly persistence layer for distributed and decentralized model persistence. Decentralized model persistence is useful in scenarios where multiple clients may access models when performing distributed computing. NeoEMF/HBase is built on top of HBase, a distributed, scalable, versioned and non-relationals big data store, specially designed to run together with Apache Hadoop.

NeoEMF/HBase takes advantage of the HBase properties by using a simple data model that minimizes data dependencies among stored objects. More specifically, NeoEMF/HBase exploits the row-locking mechanisms of HBase to provide limited ACID properties without requiring the use of transactions, which may increase latency in model operations. NeoEMF/HBase provides ACID properties at the object level, and guarantees that: (i) object queries always return the last valid state of an object; (ii) attribute modifications always succeed and produce a consistent model; and (iii) modifications of references which make the model grow monotonically always succeed and produce a consistent model.

Previous work [12] shows that key-value stores present clear benefits for storing big models, since model operations cost remains constant when models size grows. However, NeoEMF/HBase still lacks of a thorough performance evaluation. Hence, immediate future work is focused in the development of an evaluation benchmark. In this sense, we pursue to determine how the latency introduced by HBase – specially on write operations – affects the overall performance.

Additionally, a more advanced locking mechanism allowing arbitrary object locks will be implemented. Such a mechanism will provide multi-object ACID properties to the framework, allowing client applications to implement the synchronization logic to perform arbitrary, distributed and concurrent modifications.

Acknowledgments

This work is partially supported by the MONDO (EU ICT-611125) project.

References

1. CDO DB Store (2014), http://wiki.eclipse.org/CDO/DB_Store
2. CDO Hibernate Store (2014), http://wiki.eclipse.org/CDO/Hibernate_Store
3. CDO Model Repository (2014), http://www.eclipse.org/cdo/
4. CDO MongoDB Store (2014), http://wiki.eclipse.org/CDO/MongoDB_Store
5. CDO Objectivity Store (2014), http://wiki.eclipse.org/CDO/Objectivity_Store
6. Eclipse Modeling Framework (2014), http://www.eclipse.org/modeling/emf/
7. Hadoop Distributed File System (2015), http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
8. Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., Launay, D.: Neo4EMF, A Scalable Persistence Layer for EMF Models. In: Modelling Foundations and Applications, Lecture Notes in Computer Science, vol. 8569, pp. 230–241. Springer (2014)
9. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7. pp. 15–15. OSDI '06, USENIX Association, Berkeley, CA, USA (2006), http://dl.acm.org/citation.cfm?id=1267308.1267323
10. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Commun. ACM. vol. 51, pp. 107–113. ACM, NY, USA (2008)
11. Garrison, G., Kim, S., Wakefield, R.L.: Success factors for deploying cloud computing. Commun. ACM 55(9), 62–68 (Sep 2012)
12. Gómez, A., Tisi, M., Sunyé, G., Cabot, J.: Map-based transparent persistence for very large models. In: Egyed, A., Schaefer, I. (eds.) Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol. 9033, pp. 19–34. Springer Berlin Heidelberg (2015), http://dx.doi.org/10.1007/978-3-662-46675-9_2
13. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. 15(4), 287–317 (Dec 1983), http://doi.acm.org/10.1145/289.291
14. Khurana, A.: Introduction to HBase Schema Design. ;login: The Usenix Magazine 37(5), 29–36 (2012), https://www.usenix.org/publications/login/october-2012-volume-37-number-5/introduction-hbase-schema-design
15. Markus Scheidgen: EMF fragments (2014), https://github.com/markus1978/emf-fragments/wiki
16. Scheidgen, M., Zubow, A.: Map/Reduce on EMF Models. In: Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and CLoud Computing. pp. 7:1–7:5. MDHPCL '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2446224.2446231
17. The Apache Software Foundation: Apache Hadoop (2015), http://hadoop.apache.org/
18. The Apache Software Foundation: Apache HBase (2015), http://hbase.apache.org/
19. The Apache Software Foundation: Apache ZooKeeper (2015), https://zookeeper.apache.org/