This is a preprint of the article: "A. Corbellini, C. Mateos, D. Godoy, A. Zunino, S. Schiaffino: "An Architecture and Platform for Developing Distributed Recommendation Algorithms on Large-Scale Social Networks". Journal of Information Science. SAGE. 2015. ISSN 0165-5515."

The final publication is available at http://dx.doi.org/10.1177/0165551515588669

An Architecture and Platform for Developing Distributed Recommendation Algorithms on Large-Scale Social Networks

Alejandro Corbellini
ISISTAN Research Institute, UNICEN University, Argentina

Cristian Mateos
ISISTAN Research Institute, UNICEN University, Argentina

Daniela Godoy
ISISTAN Research Institute, UNICEN University, Argentina

Alejandro Zunino
ISISTAN Research Institute, UNICEN University, Argentina

Silvia Schiaffino
ISISTAN Research Institute, UNICEN University, Argentina

Abstract
The creation of new and better recommendation algorithms for social networks is currently receiving much attention due to the increasing need of new tools for assisting users. The volume of available social data as well as experimental datasets forces recommendation algorithms to scale to many computers. Given that social networks can be modelled as graphs, a distributed graph-oriented support able to exploit computer clusters arises as a necessity. In this work, we propose an architecture, called Lightweight-Massive Graph Processing Architecture (LMGPA), which simplifies the design of graph-based recommendation algorithms on clusters of computers, and a Java implementation for this architecture composed of two parts: Graphly, an API offering operations to access graphs, and jLiME, a framework that supports the distribution of algorithm code and graph data. The motivation behind the creation of this architecture is to allow users to define recommendation algorithms through the API and then customize their execution using job distribution strategies, without modifying the original algorithm. Thus, algorithms can be programmed and evaluated without the burden of thinking about distribution and parallel concerns, while still supporting environment-level tuning of the distributed execution. To validate the proposal, the current implementation of the architecture was tested using a followee recommendation algorithm for Twitter as case study. These experiments illustrate the graph API, quantitatively evaluate different job distribution strategies w.r.t. recommendation time and resource usage, and evidence the importance of providing non-invasive tuning for recommendation algorithms.

Keywords
Recommendation Algorithms, Social Networks, Large Scale Processing, Graph Databases, Graph Processing Frameworks, Work Scheduling

1. Introduction

In recent years, there has been considerable interest in the design and development of recommendation algorithms for assisting users to cope with the exponential growth of online social networks. Most experimental recommendation algorithms, particularly those developed in academia, are implemented as single-machine, single-threaded applications [67, 33, 18, 65, 66]. Even the well-known Twitter's WhoToFollow algorithm is implemented in a single-machine for reducing architectural complexity [47]. These implementations face scalability issues when the size of the underlying social graph increases, as it usually happens in today's social networks, populated with millions of users. Likewise, in situations where computing resources are scarce, generating recommendations becomes challenging. An alternative is to migrate the recommendation algorithm and/or the underlying graph database to a distributed and parallel platform that supports the execution of the algorithm on a cluster of computers (a networked array of

computers). In this context, the natural choice to handle social data is graph databases and frameworks for processing graph-based algorithms [2].

Distributed graph storage and processing systems have acquired a great interest from companies and academic institutions that handle big amounts of interconnected data such as social or geo-spatial data. For example, Facebook's social network has 801 million daily active users that upload more than 500 TB to Facebook's data stores, which are analysed in real time to harvest usage statistics. As of 2013, Twitter's 100 million users daily generated 500 million tweets per day, which are processed to show trending topics and produce targeted advertising. Moreover, the amount of data stored and queried by this kind of applications makes traditional databases and even common NoSQL solutions unfeasible to cope with the needed performance levels. Even acquiring statistically significant samples from real-world graphs, like those formed by Twitter and Facebook social networks, has proven to be challenging [25]. In the literature, the Big Data [48] term was coined to refer to large-scale data management, including its analysis and the corresponding support platforms.

In response to this challenge, some developments in the form of graph-specific databases or frameworks for processing graph algorithms have arisen. From a system design point of view, graph databases operate at the data storage level. One example are graph NoSQL databases [55], which are optimized for storing large-scale graphs. Complementary, graph processing frameworks provide programming facilities for developing graph-based algorithms. Some examples of these frameworks include Pregel [17], HipG [13], Piccolo [53] and GraphLab [68]. Surprisingly, many of these frameworks (e.g., [13, 68]) do not even cleanly support permanent graph data distributed storage, which prevent them from efficiently supporting the evolving nature of graph data in real-world social applications. One extreme case is HipG, which requires populating the main memory of the computing cluster with the graph data from a single computational node prior to processing. Another shortcoming lays on the definition of tuning code in order to adjust algorithm execution to the underlying hardware and storage characteristics, which is common on small setups of heterogeneous hardware. Most frameworks (Pregel, HipG, Piccolo) do not address this problem as they focus on algorithmic modelling rather than code distribution. To solve this problem, the developer might decide to drop the graph framework solution and use a pure distributed processing framework --e.g., MapReduce [24], BSP (Bulk Synchronous Parallel) [36], or ForkJoin [11]-- manually handling the mapping of jobs to computing nodes according to a fixed criteria. However, this approach entangles algorithm code with distributed execution concerns, which is known to cause code maintainability issues [10].

Therefore, in our view, there is a need for a new support that takes advantage of distributed data stores and provides abstractions to (optionally) allow users to exploit low-level tuning execution mechanisms, while additionally offering a simple graph traversal API tailored to recommendation algorithms. In this work, we present an architecture for distributed graph processing systems as well as its corresponding implementation, to aid the development of tunable recommendation algorithms. The architecture, called Lightweight-Massive Graph Processing Architecture (LMGPA), divides the responsibilities of a distributed graph processing system into a number of interchangeable modules. Our implementation, based on LMGPA, comprises two software modules: Graphly and jLiME. The former provides a Graph API and implements a persistent support for stored graphs. The latter is responsible for distributing and processing jobs, plus providing middleware-level services such as node discovery and failure tolerance. Moreover, jLiME supports high-level job execution models (e.g., ForkJoin, BSP, and MapReduce) that the upper layers (e.g., Graphly) can use to distribute jobs.

On one hand, when creating a recommendation algorithm, the graph API that hides all the mechanisms involved in querying the graph is vital for achieving code portability and modifiability. On the other hand, in some scenarios the developer must customize the inner mechanisms of code distribution depending on the algorithm being developed. For example, the developer might want to distribute jobs depending on the data layout (i.e., the way data is divided and replicated through the cluster) or the computational nodes capabilities. In most graph distributed processing frameworks, this involves adjusting the original algorithm or the data layout to the new requirements, which requires some effort or might be impractical. In response, we propose non-invasive job mapping strategies to customize the way job processing is distributed but without modifying the developer's original algorithm. In essence, these strategies provide a bridge between the algorithms expressed using the Graphly API and the underlying computing cluster and data layout.

This paper is organized as follows. Section 2 summarizes relevant related work on graph databases and graph processing frameworks. Section 3 describes the architecture behind Graphly and jLiME. Section 4 provides a description of the underlying distributed processing support. Section 5 introduces Graphly, its API and provides usage examples. Section 6 describes the case study used to show the importance of customizing job distribution and the experiments of the different strategies used. This case study involves expressing an existing recommendation algorithm [42, 41] using Graphly and executing it with real Twitter data. Finally, Section 7 draws some conclusions and list future work.

2. Related Work

The architecture proposed stems from two existing types of systems: graph processing frameworks and graph databases. Several frameworks that support graph processing algorithms in clusters of computers can be found in the literature. Many of those efforts are based on the well-known Bulk Synchronous Parallel (BSP) model by Valiant et. al. [36], which divides the execution of an algorithm in multiple supersteps synchronized by logical barriers in which message passing takes place. As an example, Pregel [17] is one of the most popular BSP-based computational models for graph processing. It introduces an in-memory, vertex-centric approach for modelling graph algorithms. Moreover, there are several open-source, publicly available implementations based on Pregel, such as Giraph[1], jPregel[2], GoldenOrb[3] and Phoebus[4]. Trinity [9] is another graph processing framework, created by Microsoft providing vertex-centric computation using BSP, and also graph querying capabilities and message passing between computing nodes. HipG [13] uses the notion of synchronizers to model algorithms and nodes to perform vertex-centric computations. Each synchronizer might asynchronously call many nodes and then use a barrier to wait for node execution finalization. Synchronizers might spawn multiple child synchronizers, effectively building a hierarchy of parallel synchronizers.

GraphX [49] is a BSP-based framework that covers the whole graph processing pipeline, i.e., data acquisition, graph partitioning and subsequent processing. It also provides a table view of graphs that allows the user to perform filter, map and join operations. Piccolo [53] uses a distributed key-value store to keep the data (e.g., user profiles) associated to each vertex. The map is partitioned and processed in parallel by a kernel function that generates mutations to the map. After each iteration the mutations are applied to the distributed map through the use of synchronous barriers, similarly to BSP. Moreover, Distributed GraphLab [68] bases upon an asynchronous vertex-centric computation model, where the data and graph modifications are scheduled using a distributed locking mechanism. Thus, GraphLab avoids the penalty imposed by synchronization barriers from the BSP model.

Graph processing frameworks usually load data from a persistent store and then construct a distributed in-memory representation of the graph. In many clusters, RAM memory is scarce and imposes a hard limit on the size of the graph to be processed. Additionally, many graph algorithms [42, 47, 52, 28, 35] require in-memory auxiliary data structures to handle intermediate results, which further limits the amount of memory that can be assigned to graph data. Driven by this situation, we created a graph database that support graph traversal operations and provides a model for designing distributed graph algorithms over its data. Graph data is kept in a persistent graph database and algorithm execution can be non-invasively adjusted according to different criteria, such as main memory usage, CPU speed, CPU usage, secondary disk usage, vertex storage location, among others.

In the literature regarding distributed graph databases, there are at least two types of stores: RDF (Resource Description Framework) stores and Property Graphs. RDF Stores, originated with the Semantic Web [30, 46, 54, 60] and most of them have been developed for years. The RDF specification provides the <Subject, Predicate, Object> representation of relationships in a graph of entities. The Subject is the origin vertex, the Predicate is the edge label, and the Object is the target vertex. Usually, a Context is added to express where a triple comes from. In the context of graphs, the Context is the graph where a triple belongs to. The Subject, Predicate, Object, Context (SPOC) representation is usually called a Quad. RDF Stores usually persist the SPOC tuples into a key-value store, and create several indexes to support different types of query access. For example, YARS2 is a distributed RDF Store that creates a SPOC index, and four additional indexes to support alternate access patterns. Bigdata [54] is based on concepts introduced by YARS2 [4], but adds an inference engine and an implementation of the Gather-Apply-Scatter (GAS) graph processing model. Virtuoso [46] is another example of a mature database that supports the RDF format and provides an inference mechanism over its tuples.

Property Graphs are a type of graph store where edges and vertices are modelled with an arbitrary number of properties, usually, in a key-value format. A common technique to store a property graph is to separate the different elements of a graph: vertex adjacency lists (vertices or edges), vertex data, edge input vertices, edge output vertices and edge data. Implementations of this type of databases are fairly new and are mostly related to the NoSQL movement. As an example, the Titan [58] graph database lets the user select the actual database storage support from a number of lower-level data stores, including HBase [57], Cassandra [56], BerkeleyDB [44] and Hazelcast [22]. Similarly to many new databases, Titan implements the Blueprints [61] interface, which connects it to a number of already implemented tools for graph querying and processing. Other databases provide a graph interface over a Document-oriented store (i.e., a key-value store where values have a known but flexible format, such as XML or JSON), which is the case of OrientDB [45] and ArangoDB [6]. As a final example, FlockDB [64] is an adjacency graph store that, as its name indicates, only stores the structure of the graph. It was developed at Twitter to store the followee/follower relationships of its users. All in all, our structural graph database is based on a simple distributed key-value store, where each key represents a vertex ID, and each value contains the associated vertex data. The vertex ID is used to distribute the values across the computing nodes in a cluster. Then, a Graph API is built over the key-value store to provide graph operations, e.g., obtaining the outcoming and incoming edges of a node.

In this work, we propose an architecture for graph storage and processing that, among other features, provides an API to implement traversal algorithms, i.e., algorithms that walk through the graph and might aggregate results, generating sums or counts of elements. Unlike the APIs proposed by other frameworks and databases, we focus on the problem of customizing the distribution of the graph algorithm on heterogeneous clusters. This involves providing access to

computing nodes and cluster information such as memory usage, CPU usage and job queues size, and access to information about the storage support such as data location (e.g., in which node an adjacency list resides) and cached data location. Thus, using this information, we provide predefined job mapping strategies to increase algorithm performance. Moreover, an extension mechanism is provided for further customization through ad-hoc strategies.

Indeed, job scheduling or job mapping is a well-studied problem in distributed computing [20]. On heterogeneous clusters, i.e., clusters of computers with different capabilities (main memory, CPU speed, number of CPU cores, storage capacity), the allocation of jobs to computing nodes allows distributed and parallel applications to maximize a given performance criterion [29]. In our case, when the recommendation algorithm queries a set of vertices, a user-defined mapping strategy groups those vertices into tasks and maps each task to a set of computing nodes. As an example, a locality-aware mapping strategy might group vertices by their location in the storage support.

The proposed architecture is intended to ease the development of recommendation algorithms over social networks, particularly those requiring complex navigation patterns of the social graph. This category includes link prediction problems, which compute which links are more likely to appear in the future on the graph [39]. Depending on the type of graph, such prediction might include friends, followees, products, co-workers, researchers, Web pages, and so on. There are several algorithms that use the structural information of the graph to make a prediction. Typically, these algorithms compute the similarity between nodes based on their neighbourhoods or ensembles of paths using local or global indices [38, 37, 40]. More advanced algorithms involve complex graph traversal operations. For example, PageRank [35] is a well-known algorithm that ranks Web pages on the entire Web hypergraph using the “random surfer” notion, i.e., a Web user that clicks randomly on HTML links of arbitrary Web pages. Hence, PageRank ranks Web pages by the probability that a user will be at a given Web page at any time. Personalized PageRank is a variation of the original algorithm that computes rankings based on a user’s preferences. Similarly, HITS [28] assigns an “authority” value and a “hub” value to each vertex of the graph using the number of outgoing edges and the number of incoming edges, respectively. HITS was originally used to rank Web pages resulting from a Web search, but later adapted to perform recommendations. SALSA [52] is an algorithm that uses the “hub” and “authority” notion of HITS, and the “random surfer” model from PageRank. Both SALSA and HITS receive an initial group of results from a topic search to build a ranking. More recently, the Who To Follow [47] user recommendation algorithm developed by Twitter uses Personalized PageRank to build the initial group of results, and then uses that group as an input to a SALSA-based recommendation algorithm. Other examples of followee and social recommendation algorithms in the same line are [21, 27, 31, 59, 12, 69, 26]. This type of algorithms rely on an efficient exploration of the underlying graph for generating recommendations and, consequently, solutions optimized for this domain become critical.

3. Lightweight Massive Graph Processing Architecture

In this section we describe LMGPA (Lightweight Massive Graph Analysis Architecture), a reference architecture for developing systems supporting efficient graph-traversal algorithms over large-scale graphs. Examples of such systems include not only social recommendation systems (recommendations on large social networks), but also geolocation-based systems (supporting transportation systems on large cities), bioinformatics systems (processing of large DNA sequence graphs), security audit systems (discovering attack patterns on networking logs), among others. Moreover, the proposed architecture focuses on systems that require an easy deployment and fast user adoption.

LMGPA, depicted in Figure 1, follows the Layers pattern [15] in which each layer exposes a set of operations through an API and relies on the API of the lower layer to implement those operations. User applications communicate with the Graph Abstraction layer, the topmost layer of the architecture.

This is a preprint of the article: "A. Corbellini, C. Mateos, D. Godoy, A. Zunino, S. Schiaffino: "An Architecture and Platform for Developing Distributed Recommendation Algorithms on Large-Scale Social Networks". Journal of Information Science. SAGE. 2015. ISSN 0165-5515."

The final publication is available at http://dx.doi.org/10.1177/0165551515588669

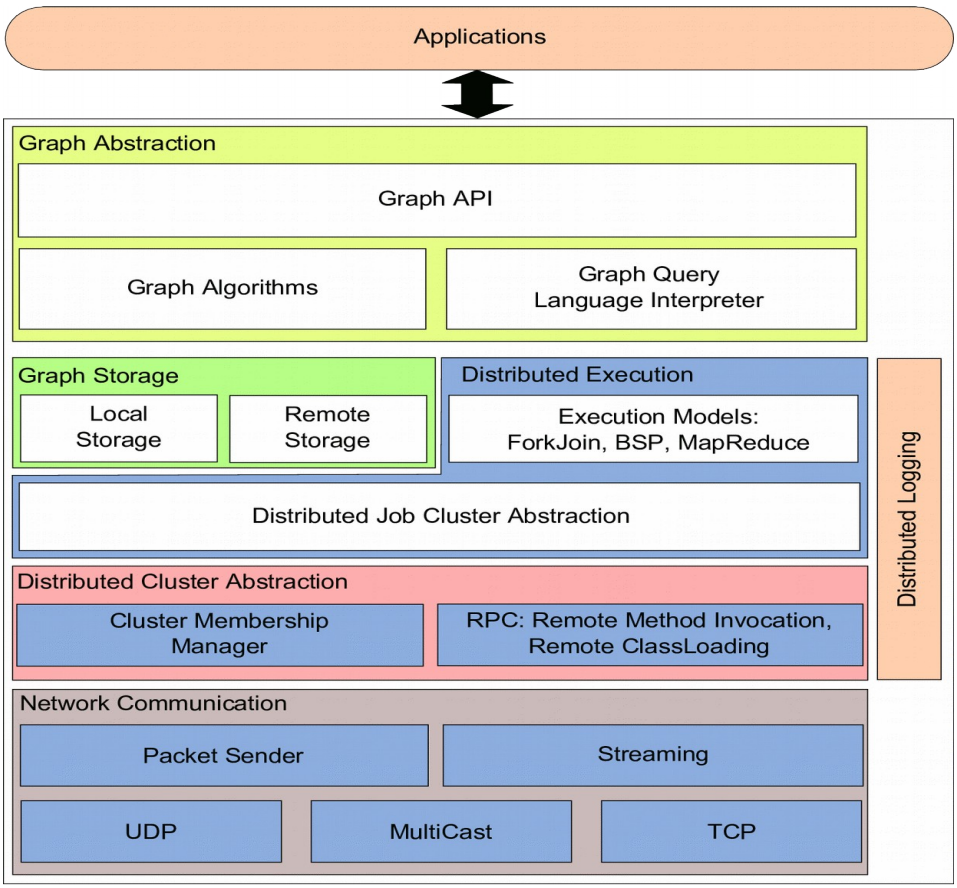


Figure 1. LMGPA layered view.

The Graph Abstraction layer defines the API to be used by the applications. This layer might be used through a declarative Graph Query Language (GQL) or a set of functions implemented in a given imperative language. Examples of GQL are Gremlin [62], Graph QL [23], and SPARQL [14]. On the other hand, as an example of standard graph APIs for imperative languages, the Blueprints [61] specification for Java provides a set of conventions to access a graph, similarly to a JDBC (Java Database Connectivity) for graphs. Additionally, this layer can implement a set of basic graph algorithms like breadth-first search, depth-first search, Floyd's algorithm and Dijkstra's algorithm. Moreover, this layer can implement primitives which underlie many recommendation algorithms such as random walks and circle of trust [47].

The second layer is the Graph Storage layer, which provides temporal or persistent storage support for the Graph API. It may be implemented using a purely graph-oriented storage like Neo4J [43] or it can provide a way to store the graph structure in a less direct way like a relational database or a key-value store. If preferred, the distributed support for this layer may be implemented using the Distributed Execution layer explained below.

The third layer corresponds to the Distributed Execution layer. This layer hides concerns related to distributed programming and provides an API to help the Graph layer and the Storage layer to distribute data and code across a cluster of computational nodes. The Distributed Execution layer provides an API to remotely execute any code using the Job abstraction. A Job is an object that can be serialized (represented as a plain stream of bytes) and executed remotely on other node. To support this abstraction, the Distributed Execution layer must implement Job managing capabilities which include, for example, job synchronization, data sharing, failure clean-up, and asynchronous/synchronous responses. Additionally, it can provide simple execution model implementations like fork-join or more advanced models like MapReduce or BSP.

The fourth layer corresponds to the Distributed Cluster Abstraction layer, which provides a view of the underlying networked LMGPA processes as a set of Peers that belong to the same Cluster. To this end, this layer implements a Cluster membership management protocol and a communication mechanism for Peers using RPC (Remote Procedure Call) which, in turn, involves handling data marshalling and unmarshalling. To implement such support, there are many existing mature RPC frameworks that can be employed, such as Protocol Buffers [34], Apache Thrift [5], RMI [63] or RabbitMQ [3].

Finally, the lowest layer is Network Communication, which implements networking functionality for sending data and code to remote processes. It must provide packet sending through TCP, UDP and UDP Multicast as well as data

streaming. Among other things, this layer must support interface disconnections and network address changes, manage TCP connections, and identify nodes by ID rather than using a hardware address. Streaming capabilities are crucial for applications where the information being transmitted cannot be handled as a whole in physical memory but, instead, can be processed by chunks in an “on demand” fashion. The Network Communication layer may be already implemented in the RPC framework used.

4. jLiME: A lightweight distributed execution framework for Java

The LMGPA architecture provides the basis for our own support for graph-based processing applications. This support is implemented via two main modules as shown in Figure 2: the Graphly module and the jLiME module. Graphly implements the Graph Abstraction layer and the storage support for representing graphs. jLiME implements the Distributed Execution layer, the Distributed Cluster Abstraction layer and the Network Communication layer. In this Section we will introduce jLiME from a bottom-up perspective.

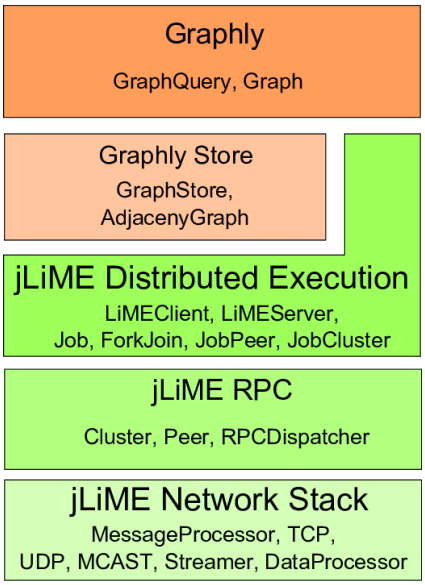


Figure 2. jLiME: A materialization of LMGPA.

4.1. jLiME Network Stack

The bottom part of the framework is the jLiME Network Stack, a reusable and flexible network communication framework, based upon the JGroups library [8]. JGroups lets the developer build a stack of modules that process incoming and outcoming network packets (e.g., modules that detect failures, acknowledge received messages, send and receive packets using UDP or TCP, and so on). The JLiME Network Stack not only allows creating stacks of packet processing modules, but also allows streaming data between Java processes. In this context, packets are arrays of bytes that are sent over the network as a whole. Streams, however, do not have a fixed size, allowing a sender to send arbitrary amounts of bytes and the receiver to consume the bytes as they arrive. In Appendix A this layer is further discussed.

4.2. jLiME RPC

jLiME provides a Remote Procedure Call module to easily perform remote calls to objects, greatly simplifying the design of distributed programs. Firstly, it introduces the concept of Peer. A Peer is a jLiME process that was detected using any discovery method provided by the network layer. Then, Peers are grouped in a Cluster object that allows the developer to select them and keep track of their state.

An RPCDispatcher object allows calling a method on a given Peer and a given registered object. jLiME also allows creating, for a given object interface, a set of helper classes to call methods from that interface as if they were local invocations. One of the helper classes is a factory class for creating a proxy object for a given Peer and a given RPCDispatcher. The usual workflow for using jLiME RPC is shown in Figure 3.

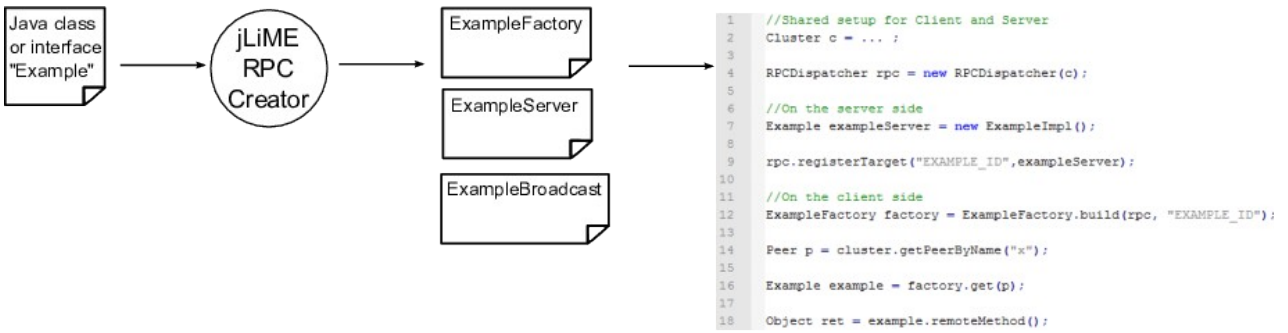


Figure 3. jLiME RPC Workflow.

Our Distributed Execution module (Section 4.3) is based on the RPC module. In this case, the RPC module simplifies sending and receiving Job objects to be executed. The Distributed Execution module registers a JobExecutor object on its RPCDispatcher, and then calls the "execute" method on remote JobExecutors using a proxy object, previously generated using the RPC module. Thus, the JobExecutor code contains a minimum amount of code related to remote execution because remote method invocations look like local invocations.

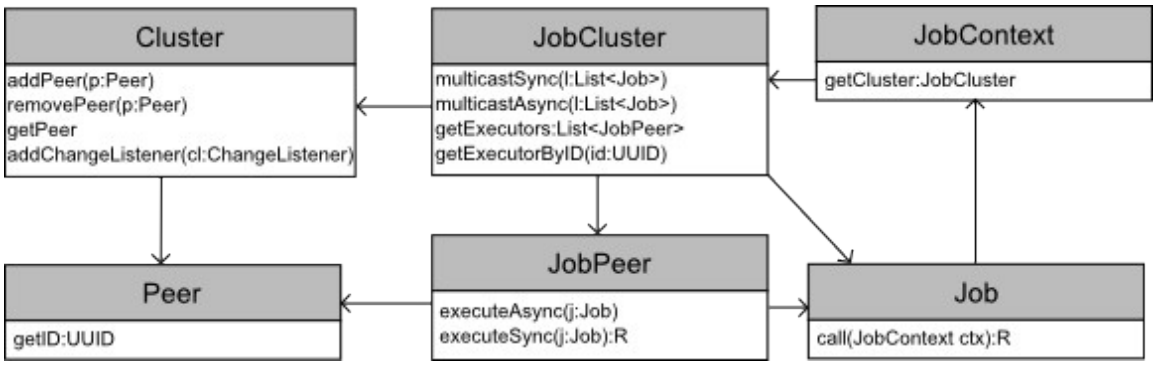


Figure 4. jLiME design: Main abstractions.

4.3. jLiME Distributed Execution

The jLiME Distributed Execution module implements the interfaces shown in Figure 4, needed to create a remote job execution environment. The first abstraction introduced by this module is the Job class, which allows a developer to execute arbitrary code on a remote machine. In order for jLiME to run a given Job on a Peer, the developer must obtain a JobPeer object, which encapsulates a Peer instance, and call the execute Job method. A JobCluster object encapsulates a Cluster object and provides access to JobPeer objects. Besides keeping track of JobPeers, the JobCluster divides peers into job executors, that allow the execution of Job objects, and job clients, that submit Jobs to job executors.

5. Graphly: A simple Java API for graph structure traversal

The graph support developed in this work provides a simple interface to traverse the structure of a graph. A graph *traversal* is an operation that follows the links of a graph, gathering information about links or vertices. We support this operation by relying on two modules: jLiME and the Graphly Store. jLiME helps to distribute any calculation needed by the Graphly API. The latter provides a distributed persistent store for the graph structure. In the following sections we will give an insight on the implementation of the Graphly Store and the Graphly API.

5.1. Graphly Store

Graphly's Store persists the graph data on a cluster of computing nodes. The implementation proposed in this work uses a distributed key-value store to save the adjacency lists of the graph, i.e., every vertex is represented by an ID, which is used as a key, and has an associated value comprised of a list of incoming or outcoming vertex IDs. Then, a modulo-based hash function to the vertex ID is used to distribute the adjacency lists among computing nodes. Each computing

node is responsible for storing a group of adjacency lists on its local database. In our implementation we used LevelDB [16] as the local database implementation. We chose LevelDB because it is a key-value store, which matches our representation of the data, and provides very fast random and sequential access.

5.2. Graphly Traverse API

Graphly provides a querying API for accessing the structure of a graph and exploring it. The class diagram representing the different graph queries is shown in Figure 5. The GraphQuery class provides a common interface for all graph queries.

The most important type of query in our design is the ListQuery, which represents a query that gives as a result a list of vertices (e.g., integer IDs). As a consequence, the most trivial query is the VertexQuery class, that already contains a list of vertices and, when executed, it only returns the contained list. This class is used to create more powerful queries and maintain homogeneity with the rest of the GraphQuery hierarchy. The *UnionQuery* and *IntersectQuery* classes perform a union or an intersection of executing two *ListQuery* queries. Finally, the *EdgeQuery* obtains the list of vertices that has incoming or outcoming edges to the currently contained *ListQuery* from the Graphly Store. As it can be

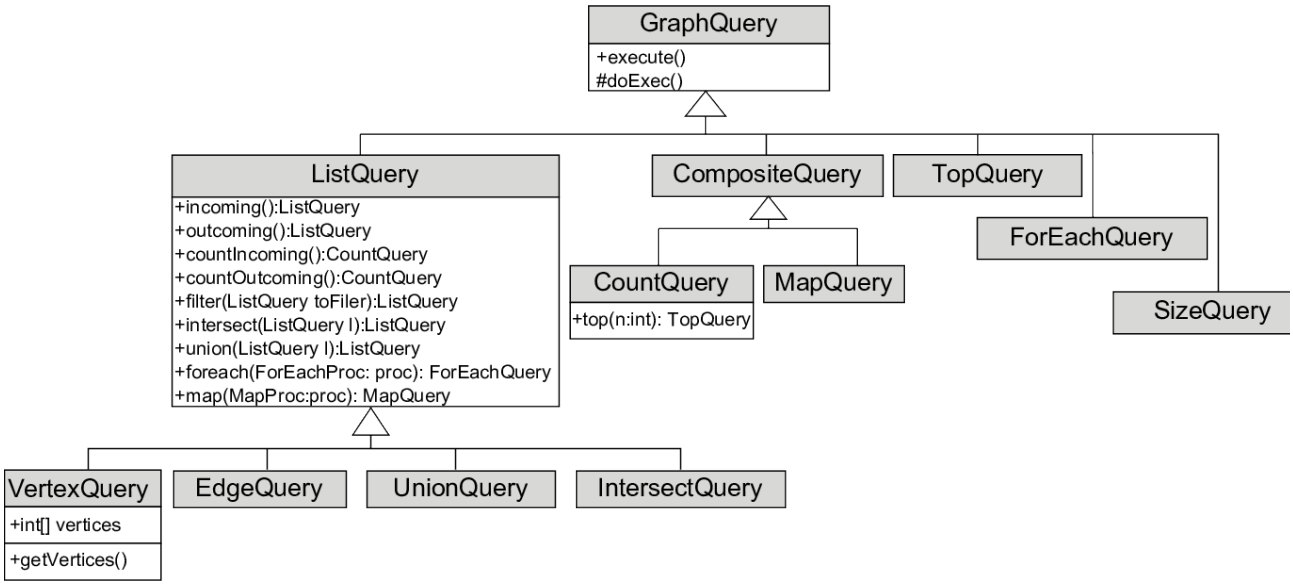


Figure 5. Graph Query hierarchy.

seen in the Figure, the *ListQuery* abstract class provides methods to create several types of queries from a given *ListQuery* object, allowing the user to chain different types of *ListQuery*.

Other types of queries perform different operations on *ListQueries*. The *SizeQuery* obtains the size of a result from a *ListQuery*, instead of transmitting the list of elements to the client. The *CountQuery* counts the number of appearances of a given vertex in a set of results. The *TopQuery* obtains the Top N elements from the result of a *CountQuery*. The *ForEachQuery* provides an extension point for a user to create custom queries to be executed on the result of a *ListQuery*. Similarly, the *MapQuery* provides a ForkJoin-like mechanism to distribute a given query along the cluster, thus balancing the processing load of the query.

A developer may define different graph traversals algorithms using the interfaces described above. As an example, the Common Neighbours metric [39] was implemented using Graphly's API. Common Neighbours is one of the simplest metrics to measure the similarity of two vertices on a graph and it is a common strategy for link prediction in recommendation systems. For two vertices x and y, let $\Gamma(x)$ denote the set of neighbours of x, their common neighbourhood is defined as $\Gamma(x) \cap \Gamma(y)$ ∨. Graphly's version of Common Neighbours is:

```
int common_neighbours = graph.vertex(x).neighbours()
                        .intersect(graph.vertex(y).neighbours()).size();
```

The *graph* variable is an instance of Graphly. Using the *getVertex* method, the user obtains a VertexQuery instance for the vertex x. The *neighbours* method creates an *EdgeQuery* that obtains the vertices having incoming or

outcoming edges to x from the Graphly Store. Finally, the result is intersected with the neighbours of vertex y, and its size is returned.

5.3. Graphly Job Mapping Strategies

Before each traversal step, it is possible to decide where to perform the merge of sub-results of such traversal. This decision depends on the user and the nature of the algorithm being developed. As an example, if the cluster consists of an heterogeneous group of computing nodes (i.e., different types of CPU, amount of RAM memory, disk storage capacity, etc.), the user may decide to distribute the merging of subresults according to each node's capabilities to achieve lower recommendation times or reducing resource usage (e.g., reducing network usage may be useful in Cloud Computing environments where inter-node communication is charged). Moreover, a CPU-bound algorithm may need to distribute tasks according to CPU capabilities, whereas a memory-intensive algorithm may distribute tasks depending on the amount of RAM memory on each node.

In any case, the algorithm original code remains unchanged. Thus, the strategies effectively work as a transparent connection between the graph algorithm and the underlying platform and storage support. A Mapping Strategy API is provided so that the user defines his/her own strategies. Nevertheless, we provide a set of predefined strategies to be used out-of-the-box. **In our experiments we used four of these strategies to show how the selection of a strategy affects an algorithm's performance and cluster resources usage. Because the algorithm is a memory and database intensive algorithm with low CPU usage we selected two memory-based strategies, a location-based strategy and as baseline for our experiments we selected a round-robin strategy. These strategies are listed below:**

- Available Memory: This strategy uses jLiME's monitoring statistics to obtain the memory currently available on each node and then divides the given list of vertices accordingly. Clearly, this strategy is dynamic, i.e., it adapts to the current status of the cluster.
- Total Memory: Similarly to the Available Memory strategy, to divide the list, the Total Memory strategy uses the maximum amount of memory that a peer can use. It is a fixed strategy that assigns more requests to nodes that have more memory available.
- Location Aware: This strategy takes advantage of the locality of the vertices. It divides the input into different lists of vertices grouped by their storage location. Such location is obtained applying a hashing function to each key. For example, let node $N1$ be responsible for vertices $a1, a3, a5$ and node $N2$ for vertices $a2, a4$. If we use a Location Aware strategy to map vertices $a1, a2, a5$, it will divide the original list into: $a1, a5 \rightarrow N1$ and $a2 \rightarrow N2$.
- Round Robin: This strategy simply divides the given list of vertices by the amount of nodes and assigns a sublist to each node. This equal division of requests among nodes makes this strategy the fairest strategy in terms of computing load. However, it does not consider either the locality of the data nor the nodes characteristics such as available memory, CPU or physical network speed. This is a job mapping strategy commonly used in distributed computing as baseline for experiments.

As mentioned before, other strategies can be defined by the user. To support this feature, Graphly provides a Mapper interface that users can extend to define their own mapping strategies. The sole function of this interface is mapping an array of vertex IDs to the available cluster nodes. Additionally, a JobContext object is provided to access the cluster information and obtain shared objects from the platform. The RoundRobin Mapper implementation is shown in Figure 6 as an example.

```
public class RoundRobinMapper extends Mapper {
    @Override
    public Map<ClientNode, List<Integer> > map(int[] data, JobContext ctx) {
        HashMap<ClientNode, TIntArrayList> div = new HashMap<ClientNode, TIntArrayList>();
        ArrayList<ClientNode> serverList = env.getCluster().getExecutors();
        int count = 0;
        for (int i : data) {
            ClientNode p = serverList.get(count);
            count = (count + 1) % serverList.size();
            div.get(p).add(i);
        }
        return div;
    }
}
```

Figure 6. Round Robin Mapping Strategy implementation.

Regarding the strategies that perform a division based on node information, the user may use the CriteriaMapper object. A CriteriaMapper divides the list of vertex IDs proportionally to a provided (quantitative) node metric such as available RAM memory, CPU speed, current amount of jobs being executed, among other metrics. For example, the Available Memory strategy uses a CriteriaMapper instantiated with a metric named "memory.available". Thus, using this strategy, the nodes that report bigger amounts for the "memory.available" metric will receive more vertex IDs than the other nodes. In our tests, we created our custom strategies using Spring Configuration [5]. The Available Memory mapping strategy configuration is shown in Figure 7.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans .>
    <bean id="availablememory"
        class="edu.jlime.collections.adjacencygraph.mappers.CriteriaMapper">
        <constructor-arg value="memory.available" />
    </bean>
</beans>
```

Figure 7. Available Memory configuration using Spring Configuration.

6. Case Study

In order to validate the proposed architecture, we implemented a recommendation algorithm and performed experiments with different job distribution strategies in the current materialization of LMGPA. The algorithm used in this paper for illustrating the importance of mapping strategies is a followee recommender algorithm for Twitter users [41]. To simplify the analysis of strategy impact, we focus our experiments on a recommendation algorithm that only uses the topological information of the graph, such as other state-of-the-art algorithms (e.g. WhoToFollow, SALSA, HITS and PageRank). However, the platform allows adding content to both vertices and edges, e.g. "tweet lists" and "type of relationship", respectively. Naturally, algorithms that process content have memory and CPU requirements that are much higher than topological algorithms.

Section 6.1 presents the algorithm definition and its adaptation to the Graphly API. Section 6.2 describes the experimental settings, including the dataset used, the hardware features of the computer cluster and the initial configuration. In turn, Section 6.3 shows how the choice of a mapping strategy affects the algorithm execution times and cluster resource usage.

6.1. Followee Recommendation Algorithm

In the context of the Twitter social network, a user has a group of followees (outgoing edges) and followers (incoming edges). The algorithm used as case study, proposed in [41], can be divided in two stages. It first explores the followee/follower network near the target user (i.e., the user receiving the suggestions) to select a set of potential users to recommend. Then, the candidates are filtered and ranked according to different criteria [42], such as the number of common friends with the target user or the matching of their content-based profiles. In this work we focus on supporting the first stage of the algorithm on a large graph, letting aside the quality of the actual recommendation.

The exploration of the followee/follower network in this algorithm is based on the characterization of Twitter users made in several studies [1, 7] and the fact that online social networks had become real-time information sources and news spreading mediums besides fostering the formation of social ties [19]. From an information-centric point of view, users are mainly divided in two categories: information sources and information seekers. Users behaving as information sources tend to collect a large amount of followers as they post useful information or news, whereas information seekers follow several users to get information but rarely post a tweet themselves.

The rationale behind the graph traversal stage of this algorithm relies in the categorization of users as information seekers or information sources. Thus, it is assumed that the target user is an information seeker that has already identified some interesting Twitter users acting as information sources (i.e., his/her followees). Other people who also follow these people (i.e. followers of the user followees) are likely to share some interests with the target user and might have discovered other relevant information sources on the same topics (i.e., their followees). This last group is then composed of potential candidates for suggesting to the user as future followees.

More formally, the search of candidate users is performed according to the following steps:

This is a preprint of the article: "A. Corbellini, C. Mateos, D. Godoy, A. Zunino, S. Schiaffino: "An Architecture and Platform for Developing Distributed Recommendation Algorithms on Large-Scale Social Networks". Journal of Information Science. SAGE. 2015. ISSN 0165-5515."

The final publication is available at http://dx.doi.org/10.1177/0165551515588669

Step 1. Starting with the target user u_T (an information seeker), obtain the list of users he/she follows into a list $S=\{s1, s2, \ldots, sn\}$. Members of S are likely to be information sources.

Step 2. For each element in S, obtain its followers. Let us call the union of all these lists L, i.e., $L=\forall s \in followees(uT)\, followers(s)$. Members of L are likely to be information seekers.

Step 3. For each element in L obtain its followees. Let us call the union of all these lists T, i.e. $T=\forall l \in L\, followees(l)$. Members of T are likely to be information sources.

Step 4. Exclude from T those users who the user being recommended is already following. Let us call the resulting list of candidates R.

Using the Graphly API, this algorithm can be expressed as depicted in Figure 8. The adapted version of the algorithm broadly contains the next steps:

(1) Get the followees (*outgoing*) of the target user.
(2) Get the followers (*incoming*) of the followees and remove the original followees from the list.
(3) Count the followees (*countOutgoing*) of the followers and remove the original followees from the map.
(4) Obtain the top n elements with most appearances ($n=10$ in this example).

```
ListQuery followees = graph.get(user).ougoing();
                      VertexQuery     GetQuery

TopQuery query = followees.incoming().remove(followees)
                 GetQuery (removing original users from the result)

.countOutgoing().remove(followees).top(10);
 CountQuery(counts users                Filters the top 10 users
 apperances and removes original users) from the count query.

Map<Integer,Integer> recommendation = query.query(cluster);
                                      Executes the whole
                                      query at a Server
```

Figure 8. The studied recommendation algorithm expressed using the Graphly API.

The *followees* object appears twice in the query because the original followees must not be part of the recommendation as they are already being followed by the user.

6.2. Experimental Setting

Experiments were carried out using a Twitter dataset[6] as of July 2009 consisting of approximately 1,400 million relationships between more than 41 million users [19]. This is a extensively study dataset, containing topological information about the social network, i.e., it has binary relationships between user IDs, but does not include tweets. This is not a limitation since the test algorithm builds recommendations based on user relationships only, searching for candidate followees in the Twitter graph. In addition, the dataset provides a good benchmark for topology-based link-prediction algorithms since is a complete, real-word graph, not a sub-sample or artificially generated one. Thus, its is possible to tune the platform and observe its behaviour with real-word processing requirements.

For testing the algorithm, we selected two test groups of users out of the complete dataset with the goal of illustrating the performance of the different strategies for varying algorithm needs. For selecting a representative test user list, we first filtered the users using the information source ratio [42], denoting to what extent a user can be considered as an information source. This ratio is defined as follows:

$$IS(u)=\frac{followers(u)}{followers(u)+followees(u)} \quad (1)$$

We set this ratio to be less or equals than 0.5, which means that the number of followees had to be equal or larger than the number of followers. This restriction arises as a natural way of selecting which users would have any interest in receiving recommendations, since the algorithm under study is designed for recommending followees to information consumers more than to information producers.

The first test group, called the "IS" group, comprises the top-100 users with the smallest IS values. From the algorithm point of view, these users are the most interesting information consumers to recommend other users to follow. To obtain the second test group, called the "Followees" group, we ordered the filtered users by number of followees and kept the top-100. These users represent the most influential users of the test set and were used to stress the computational nodes employed assuming they will produce long lists of followers and followees.

Table 1. Cluster hardware characteristics.

	First set (5 nodes)	Second set (3 nodes)
CPU	AMD Phenom II X6 1055T 2.8 Ghz	AMD FX 6100 3.3 Ghz
# of cores	6	6
RAM	8 GB	16 GB
Physical Network	1 Gbit Ethernet	1 Gbit Ethernet
Hard Disk	500 GB - 7200 RPM	500 GB - 7200 RPM

Regarding the cluster characteristics, we used a heterogeneous cluster of 8 nodes divided in two sets, each having identical hardware characteristics. Table 1 summarizes the most relevant characteristics of each set of nodes.

6.3. Results

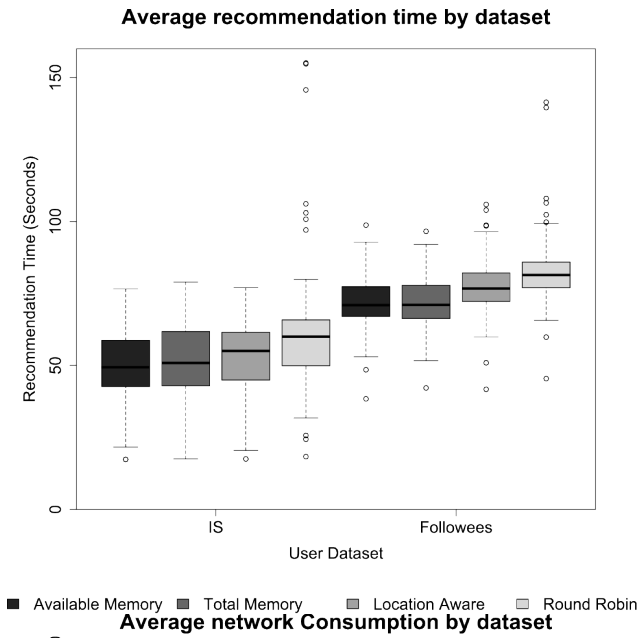
The experiments were carried out as follows. For each user in the testing set, we ran the algorithm 5 times for each one of the strategies presented in Section 5.3. In every execution, we measured the total recommendation time, the bytes sent over the network and the maximum memory consumed (the biggest memory spike). Using this information we obtained the average number of bytes sent over the network, the average maximum memory consumption and the average recommendation time. It is worth noting that the number of times we ran the algorithms yielded as a result very low statistical deviations, and thus these averages are statistically significant. The distribution of average values for a given group of users and a given strategy are summarized in Figure 9.

As it was expected, users in the "IS" group demand, on average, less time to process as well as less resources. Instead, the "Ffollowees" group was intentionally selected to stress the cluster. Users in this group need more resources to be processed and, consequently, their processing takes longer than users in the "IS" group.

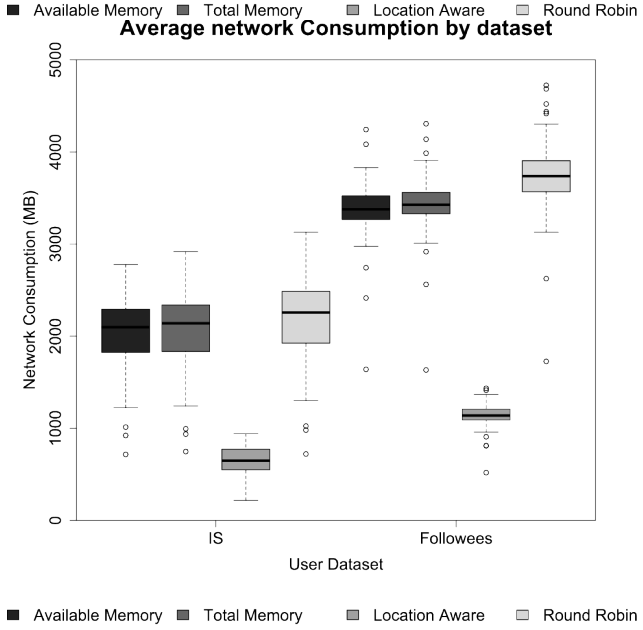
Regarding the performance of the mapping strategies for both groups of users, the Round Robin strategy provided the worst recommendation times as well as higher network and memory usage. This can be explained by the fair but random division of tasks processing.

This is a preprint of the article: "A. Corbellini, C. Mateos, D. Godoy, A. Zunino, S. Schiaffino: "An Architecture and Platform for Developing Distributed Recommendation Algorithms on Large-Scale Social Networks". Journal of Information Science. SAGE. 2015. ISSN 0165-5515."

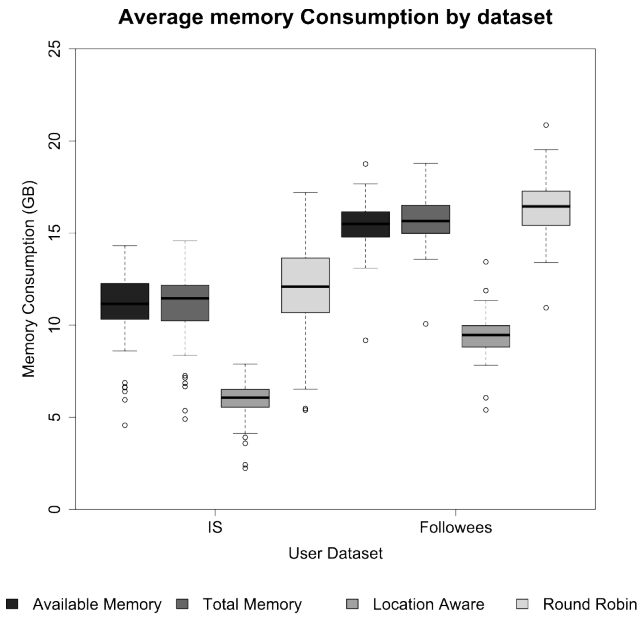
The final publication is available at http://dx.doi.org/10.1177/0165551515588669



(a)



(b)



(c)

Figure 9. Average results for each testing group of users.

The memory-based strategies provided similar performance in network and memory consumption and recommendation time. These strategies performed better than the Location Aware strategy and the Round Robin strategy. This might be explained by the memory-intensive nature of the algorithm used as case study: on each step, the algorithm generates exponentially larger sub-results that are merged and allocated as input for the next step. Thus, an allocation of tasks according to memory capabilities of the nodes provides better performance by using less virtual memory from the Operating System (also letting the OS to use more memory for disk cache) and, in the case of our implementation language, producing less garbage collection from the Java Virtual Machine.

Finally, the Location Aware strategy provided intermediate performance in terms of recommendation time. However, memory consumption was considerably lower than other strategies and, as expected, the network consumption was minimal. In scenarios where memory is scarce or network speed is an impediment, the Location Aware strategy may be a better fit than other strategies. Moreover, in some scenarios using Cloud processing platforms (e.g., Amazon AWS or Google Compute Engine), using less network bandwidth may have economic benefits.

To further compare strategies, a typical execution trace for the implemented recommendation algorithm is shown in Figure 10. In this case, the Location Aware strategy presents a memory usage pattern that is almost linear, whereas the other strategies show an exponential pattern on early steps of the recommendation algorithm. This is due to the nature of the algorithm which generates an exponential amount of remote requests on each traversal step. As expected, the network usage is similar for non-locality strategies. Moreover, for those strategies, the growth in network usage matches a growth in memory usage.

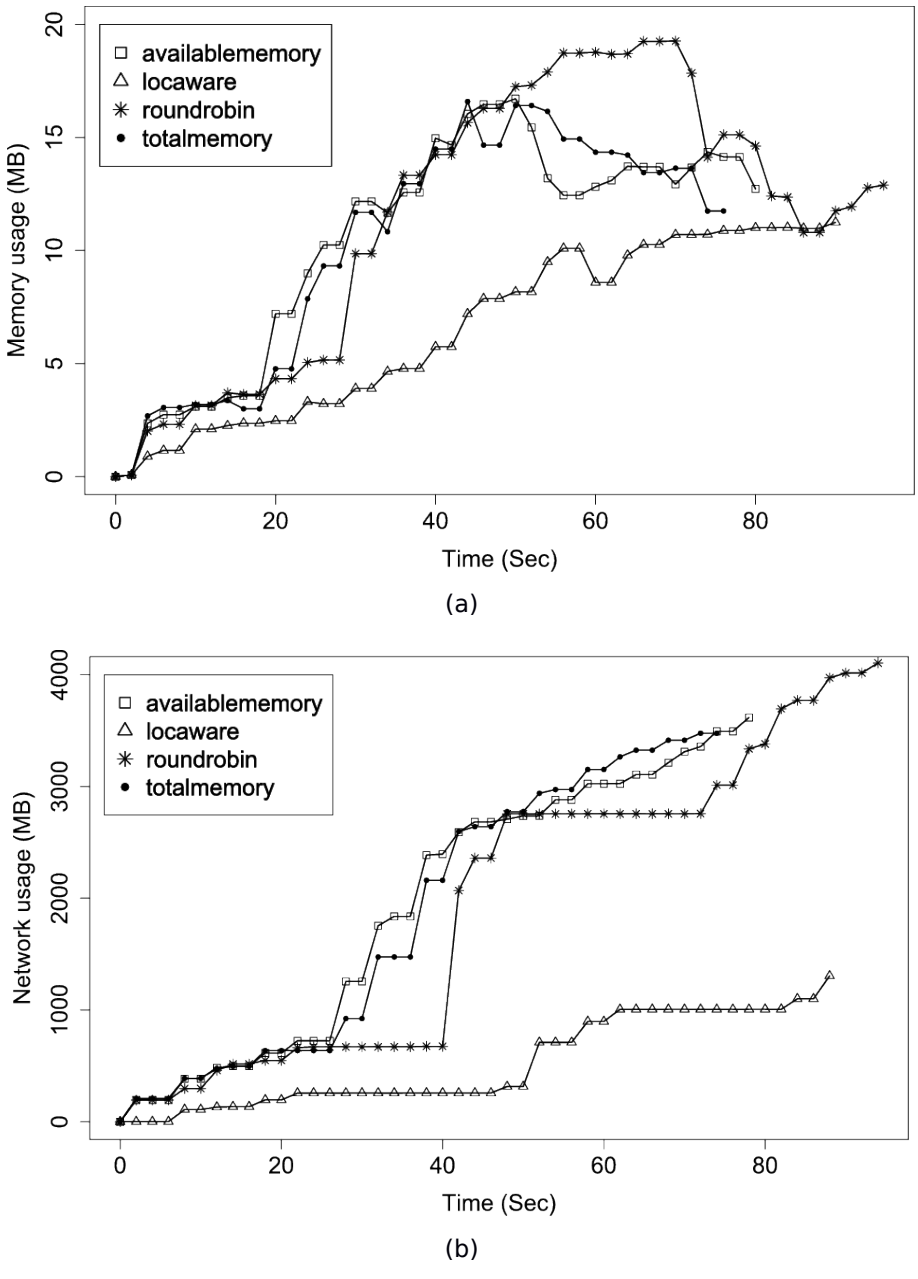


Figure 10. Cluster network and memory usage for an example user.

Table 2. Strategy suitability according to the environment, the graph and the algorithm characteristics.

Strategies	Environment Characteristics		Graph Topology		Algorithm Characteristics	
	Limited Memory	Slow/Costly Network	Natural Graphs	Low-Connected Graphs	Memory-intensive	CPU-intensive
Available Memory	Poor	Very Poor	Good	Good	Very good	Good
Total Memory	Poor	Very Poor	Good	Good	Very good	Good
Location Aware	Very Good	Very Good	Good	Good	Good	Good
Round Robin	Very Poor	Very Poor	Poor	Good	Poor	Very Good

Although the platform allows developers of recommendation algorithms to decide the strategy according to the algorithm requirements, based on the previous results it is possuble to draw some conclusions to guide this selection process. Table 2 provides assistance in the decision of which strategy shall be used depending on the environment characteristics, graph topology and algorithm characteristics. As environment characteristics we considered two scenarios: a Limited Memory scenario and a Slow/Costly Network scenario. The former corresponds to environments where the nodes have relatively low RAM memory, which is the case of, for instance, commodity clusters, or environments including embedded devices like cell phones or tablets [32] (a.k.a. mobile Grids). The latter scenario considers slow computer networks, where transference between nodes incurs in a big performance penalty or cases where network transfers are charged like in, for example, paid Cloud Computing environments. Under graph topology we only consider graphs that are highly interconnected (e.g. natural graphs) and graphs that have a low amount of edges w.r.t. their vertices. Finally, in algorithm characteristics we identify two scenarios: implementing a memory-intensive recommendation algorithm or a CPU-intensive algorithm.

The scenarios depicted are far from exhaustive, mainly because we only focused on those scenarios that are related to the case study analysed. Then, even when the criteria used to fill in part of the Table is somewhat subjective, our main goal is to guide potential jLiME users into quickly obtaining an initial, working configuration for efficiently running their recommendation algorithms. Possible values for strategy performance in a given scenario range from Very Poor to Very Good. A Very Poor value does not necessarily mean that the current strategy is unusable for that scenario, but it might not perform as well as other strategies. Additionally, these values depend on the scenario being evaluated, e.g., if we consider the Limited Memory scenario, a Very Poor value means that the strategy introduces high memory consumption, while a Very Good value means that the strategy favours low memory consumption.

The selection of a strategy for a given environment, graph and algorithm combination is straightforward. For example, in a Slow Network scenario the best choice is the Location Aware strategy. On the other hand, in a Limited Memory scenario, a Low-Connected Graph and a CPU-intensive algorithm, the best choice might be a simple Round Robin strategy. Nevertheless, because the scenarios are not mutually exclusive, conflicts between requirements may arise resulting in a trade-off situation.

7. Conclusions

In this paper, we presented a novel architecture and a corresponding implementation for designing distributed recommending algorithms. The algorithms are expressed in terms of graph traverse operations defined by the API of our graph support, Graphly. The underlying framework, jLiME, deals with networking concerns like remote code execution. The core motivation of this work is the creation of a customizable and transparent mechanism to distribute the processing of prototype recommending algorithms. This mechanism, called mapping strategies, works as a bridge between Graphly API and jLiME, by exposing information about the computational capabilities of every node, including the graph data distribution.

This work shows the importance of such mechanism by comparing several strategies and their impact in recommendation time and resource usage, especially on heterogeneous clusters. As a case study, we used an existing structural recommendation algorithm for Twitter and real Twitter data, and made recommendations to two different groups of 100 users to show the differences in recommendation times and resource usage. For this type of recommending algorithm, the memory-based strategies showed a very good recommendation time, followed by a

strategy that uses data locality. This can however vary from algorithm to algorithm, and thus the architecture stressed the need for a flexible tuning support so that users can use or even define the strategies which best suit their algorithms.

As future work, we will examine the impact of mapping strategies in algorithms implemented using a different processing model, like Pregel. Some iterative graph algorithms, such as PageRank, can be better expressed in a vertex-centric processing model rather than a Divide-and-Conquer model like MapReduce or ForkJoin, i.e., the ones which jLiME is currently based upon. Selecting a better model for a particular algorithm also results in performance benefits by reducing code and algorithmic overhead imposed by a less-fitted model.

Furthermore, we will explore new mapping strategies and job-stealing strategies to rebalance processing if nodes are overloaded. While mapping strategies decide how task distribution is performed, job stealing strategies perform a runtime adjustment on that distribution by stealing tasks that have not completed on busy nodes and allocating them on idle nodes. In other words, the current mapping strategies follow a push-based approach to distributing jobs to machines, whereas job stealing follows a pull-based approach. Indeed, these approaches have been studied in the context of other types of resource-demanding applications [50] and thus we believe they are worth to be explored.

Another extension of this work includes the definition of reusable, higher-level graph primitive operations especially suited to recommending algorithms beyond simple graph primitives, like obtaining common neighbours, PageRank calculation or obtaining the “circle of trust” -- as described by the WhoToFollow algorithm [47] -- of a given vertex. The introduction of these new primitives might also require defining higher-level mapping strategies. A strategy that divides tasks according to the amount of neighbours of a given vertex might provide better load balancing when the primitive (or the user’s algorithm) is based on processing vertices neighbourhoods.

The inclusion of algorithms that use graph content besides topology will foster the creation of new strategies that take into account text mining requirements. For efficiency reasons, in many cases recommendation algorithms provide a preliminary ranking of users based on topological information, which is given as input to a content-based recommendation algorithm to build a final ranking. This approach offers a better balance to execution time-recommendation effectiveness, but still many job mapping strategies could be explored at both ends to further speeding up recommendations. Additionally, in terms of Graphly, these algorithms may become primitives that can be reused by other algorithms.

Notes

1. Giraph Web Page, http://giraph.apache.org/
2. jPregel Web Page, http://kowshik.github.com/JPregel/
3. GoldenOrb Web Page, http://www.goldenorbos.org/
4. Phoebus Web Page, https://github.com/xslogic/phoebus
5. Spring Framework Web Page, http://spring.io/
6. http://an.kaist.ac.kr/traces/WWW2010.html

Funding

This work has been partially funded by ANPCyT, through project PICT-2011-0366, and CONICET, under grant PIP No. 114-200901-00381.

References

[1] Akshay Java, Xiaodan Song, Tim Finin, Belle Tseng. Why we Twitter: Understanding microblogging usage and communities. Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis (WebKDD/SNA-KDD '07):56—65, 2007.

[2] Alejandro Corbellini, Daniela Godoy, Cristian Mateos, Alejandro Zunino, Silvia Schiaffino. A Programming Interface and Platform Support for Developing Recommendation Algorithms on Large-Scale Social Networks. In Collaboration and Technology, volume 8658 of Lectures Notes in Computer Science: 67—74, Springer, 2014.

[3] Alvaro Videla, Jason J. W. Williams. RabbitMQ in action. Manning, 2012.

[4] Andreas Harth, Jürgen Umbrich, Aidan Hogan, Stefan Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In The Semantic Web, volume 4825 of Lectures Notes in Computer Science: 211—224, Springer, 2007.

[5] The Apache Foundation. Apache Thrift. 2014. Accessed: 19-08-2014.

[6] Arango DB. Arango DB. 2014. Accessed: 20-08-2014.

[7] Balachander Krishnamurthy, Phillipa Gill, Martin Arlitt. A few chirps about Twitter. Proceedings of the 1st Workshop on Online Social Networks (WOSP'08):19—24, 2008.

[8] Bela Ban, others. JGroups, a toolkit for reliable multicast communication. URL: http://www.jgroups.org, 2002.

[9] Bin Shao, Haixun Wang, Yatao Li. The Trinity Graph Engine. 2012.

[10] Cristian Mateos, Alejandro Zunino, Marcelo Campo. JGRIM: An approach for easy gridification of applications. Future Generation Computer Systems, 24(2):99—118, 2008.

[11] Cristian Mateos, Alejandro Zunino, Matías Hirsch. EasyFJP: Providing hybrid parallelism as a concern for divide and conquer Java applications. Computer Science and Information Systems, 10(3):1129—1163, 2013.

[12] Daniel Schall. Who to follow recommendation in large-scale online development communities. Information and Software Technology, 56(12):1543—1555, 2013.

[13] Elzbieta Krepska, Thilo Kielmann, Wan Fokkink, Henri Bal. HipG: Parallel processing of large-scale graphs. ACM SIGOPS Operating Systems Review, 45(2):3—13, 2011.
[14] Eric Prud'Hommeaux, Andy Seaborne. SPARQL query language for RDF. 2008.
[15] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Peter Sommerlad, Michael Stal. Pattern-oriented software architecture: A system of patterns. John Wiley and Sons, 1996.
[16] Google. LevelDB. 2014. Accessed: 12-09-2014.
[17] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. Proceedings of the 2010 International Conference on Management of Data (SIGMOD '10):135—146, 2010.
[18] Guillaume Durand, Nabil Belacel, François LaPlante. Graph theory based model for learning path recommendation. Information Sciences, 251:10—21, 2013.
[19] Haewoon Kwak, Changhyun Lee, Hosung Park, Sue Moon. What is Twitter, a social network or a news media?. Proceedings of the 19th International Conference on World Wide Web (WWW'10):591—600, 2010.
[20] Haluk Topcuoglu, Salim Hariri, Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems, 13(3):260—274, 2002.
[21] Hao Wu, Vikram Sorathia, Viktor K. Prasanna. Predict Whom One Will Follow: Followee Recommendation in Microblogs. Proceedings of the 2012 International Conference on Social Informatics (SocialInformatics):260—264, 2012.
[22] Hazelcast Inc. Hazelcast. 2014. Accessed: 20-08-2014.
[23] Huahai He, Ambuj K. Singh. Query Language and Access Methods for Graph Databases. In Managing and Mining Graph Data. Volume 40, Springer, 2010.
[24] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51(1):107—113, 2008.
[25] Jianguo Lu, Hao Wang. Variance reduction in large graph sampling. Information Processing & Management , 50(3):476—491, 2014.
[26] Jie Zhang, Yuan Wang, Julita Vassileva. SocConnect: A personalized social network aggregator and recommender. Information Processing & Management, 49(3):721—737, 2013.
[27] John Hannon, Kevin McCarthy, Barry Smyth. Finding useful users on twitter: Twittomender the followee recommender. In Advances in Information Retrieval, volume 6611 of Lectures Notes in Computer Science: 784—787, Springer, 2011.
[28] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. Journal of the ACM, 46(5):604—632, 1999.
[29] Jong-Kook Kim, Sameer Shivle, Howard Jay Siegel, Anthony A. Maciejewski, Tracy D. Braun, Myron Schneider, Sonja Tideman, Ramakrishna Chitta, Raheleh B. Dilmaghani, Rohit Joshi, Aditya Kaul, Ashish Sharma, Siddhartha Sripada, Praveen Vangari, Siva Sankar Yellampalli. Dynamic mapping in a heterogeneous environment with tasks having priorities and multiple deadlines. Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), 2003.
[30] José M. Morales-del-Castillo, Eduardo Peis, Antonio A. Ruiz, Enrique Herrera-Viedma. Recommending biomedical resources: A fuzzy linguistic approach based on Semantic Web. International Journal of Intelligent Systems, 25(12):1143—1157, 2010.
[31] Josh Jia-Ching Ying, Eric Hsueh-Chan Lu, Vincent S. Tseng. Followee recommendation in asymmetrical location-based social networks. Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp '12):988—995, 2012.
[32] Juan Manuel Rodriguez, Cristian Mateos, Alejandro Zunino. Energy-efficient job stealing for CPU-intensive processing in mobile devices. Computing, 96(2):1—31, 2014.
[33] Katharina Rausch, Eirini Ntoutsi, Kostas Stefanidis, Hans-Peter Kriegel. Exploring subspace clustering for recommendations. Proceedings of the 26th International Conference on Scientific and Statistical Database Management (SSDBM '14):42:1—42:4, 2014.
[34] Kenton Varda. Protocol buffers: Google's data interchange format. 2008.
[35] Lawrence Page, Sergey Brin, Rajeev Motwani, Terry Winograd. The PageRank citation ranking: Bringing order to the Web. 1999.
[36] Leslie G. Valiant. A bridging model for parallel computation. Communications of the ACM, 33(8):103—111, 1990.
[37] Longjie Li, Min Ma, Peng Lei, Mingwei Leng, Xiaoyun Chen. S2R&R2S: A framework for ranking vertex and computing vertex-pair similarity simultaneously. Journal of Information Science, 40(6):723-735, 2014. URL http://jis.sagepub.com/content/40/6/723.abstract.
[38] Longjie Li, Lvjian Qian, Jianjun Cheng, Min Ma, Xiaoyun Chen. Accurate similarity index based on the contributions of paths and end nodes for link prediction. Journal of Information Science, 2014. URL http://jis.sagepub.com/content/early/2014/12/22/0165551514560121.abstract.
[39] Linyuan Lu, Tao Zhou. Link prediction in complex networks: A survey. Physica A: Statistical Mechanics and its Applications, 390(6):1150—1170, 2011.
[40] Linyuan Lü, Ci-Hang Jin, Tao Zhou. Similarity index based on local paths for link prediction of complex networks. Phys. Rev. E, 80:046122, 2009. URL http://link.aps.org/doi/10.1103/PhysRevE.80.046122.
[41] Marcelo Armentano, Daniela Godoy, Analía Amandi. Topology-based recommendation of users in micro-blogging communities. Journal of Computer Science and Technology. Special Issue on Data Mining on Social Networks and Social Web, 27(3):624—634, 2012.
[42] Marcelo Armentano, Daniela Godoy, Analía Amandi. Towards a followee recommender system for information seeking users in Twitter. Proceedings of the International Workshop on Semantic Adaptive Social Web (SASWeb'11), 2011.
[43] Inc. Neo Technology. Neo4J. 2013. Accessed: 05-08-2013.
[44] Oracle. Oracle BerkeleyDB. 2014. Accessed: 20-08-2014.
[45] Orient Technologies. Orient DB. 2014. Accessed: 20-08-2014.
[46] Orri Erling, Ivan Mikhailov. Virtuoso: RDF support in a native RDBMS. In Semantic Web Information Management: 501—519. Springer, 2010.

[47] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, Reza Zadeh. WTF: The Who to Follow Service at Twitter. Proceedings of the 22th International World Wide Web Conference (WWW 2013):505—514, 2013.
[48] Paul Zikopoulos, Chris Eaton. Understanding big data: Analytics for enterprise class Hadoop and streaming data. McGraw-Hill Osborne Media, 2011.
[49] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica. GraphX: A resilient distributed graph system on Spark. Proceedings of the 1st International Workshop on Graph Data Management Experiences and Systems (GRADES '13):2:1—2:6, 2013.
[50] Rob V. Van Nieuwpoort, Gosia Wrzesińska, Ceriel J. H. Jacobs, Henri E. Bal. Satin: A High-level and Efficient Grid Programming Model. ACM Transactions on Programming Languages and Systems, 32(3):9:1—9:39, 2010.
[51] Ron Johnson. J2EE development frameworks. Computer, 38(1):107—110, 2005.
[52] Ronny Lempel, Shlomon Moran. SALSA: The stochastic approach for link-structure analysis. ACM Transactions on Information Systems, 19(2):131—160, 2001.
[53] Russell Power, Jinyang Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10), 10:1—14, 2010.
[54] LLC. SYSTAP. BigData. 2013.
[55] Sherif Sakr, Anna Liu, Daniel M. Batista, Mohammad Alomari. A Survey of Large Scale Data Management Approaches in Cloud Environments. IEEE Communications Surveys Tutorials, 13(3):311—336, 2011.
[56] The Apache Foundation. Cassandra. 2014. Accessed: 20-08-2014.
[57] The Apache Foundation. HBASE. 2014. Accessed: 20-08-2014.
[58] Think Aurelius. Titan. 2014. Accessed: 14-04-2014.
[59] Tianqi Chen, Linpeng Tang, Qin Liu, Diyi Yang, Saining Xie, Xuezhi Cao, Chunyang Wu, Enpeng Yao, Zhengyang Liu, Zhansheng Jiang, Cheng Chen, Weihao Kong, Yong Yu. Combining factorization model and additive forest for collaborative followee recommendation. KDD Cup Workshop, 2012.
[60] Tim Berners-Lee, James Hendler, Ora Lassila. The Semantic Web. Scientific American, 284(5):28—37, 2001.
[61] Tinkerpop. Blueprints. 2014. Accessed: 19-08-2014.
[62] Tinkerpop. Gremlim. 2014. Accessed: 19-08-2014.
[63] Troy Bryan Downing. Java RMI: Remote Method Invocation. Wiley, 1998.
[64] Twitter Inc. FlockDB. 2013. Accessed: 05-08-2013.
[65] Xiaofang Wang, Jun Ma, Ming Xu. Group Recommendation for Flickr Images by 4-order Tensor Decomposition. Journal of Computational Information Systems, 10(3):1315—1322, 2014.
[66] Xuetao Guo, Jie Lu. Intelligent e-government services with personalized recommendation techniques. International Journal of Intelligent Systems, 22(5):401—417, 2007.
[67] Yuchen Jing, Xiuzhen Zhang, Lifang Wu, Jinqiao Wang, Zemeng Feng, Dan Wang. Recommendation on Flickr by combining community user ratings and item importance. Proceedings of the IEEE International Conference on Multimedia and Expo (ICME 2014):1—6, 2014.
[68] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment, 5(8):716—727, 2012.
[69] Yung-Ming Li, Tzu-Fong Liao, Cheng-Yang Lai. A social recommender mechanism for improving knowledge sharing in online forums . Information Processing & Management , 48(5):978 - 994, 2012. Large-Scale and Distributed Systems for Information Retrieval.

Appendix A. JLiME Network Layer

The jLiME Stack packet-sending implementation is based on our Message Processor (MP) abstraction. Each MP receives a message, processes it and sends it to the next MP. On each MP, the messages are added to a queue and scheduled for local processing.

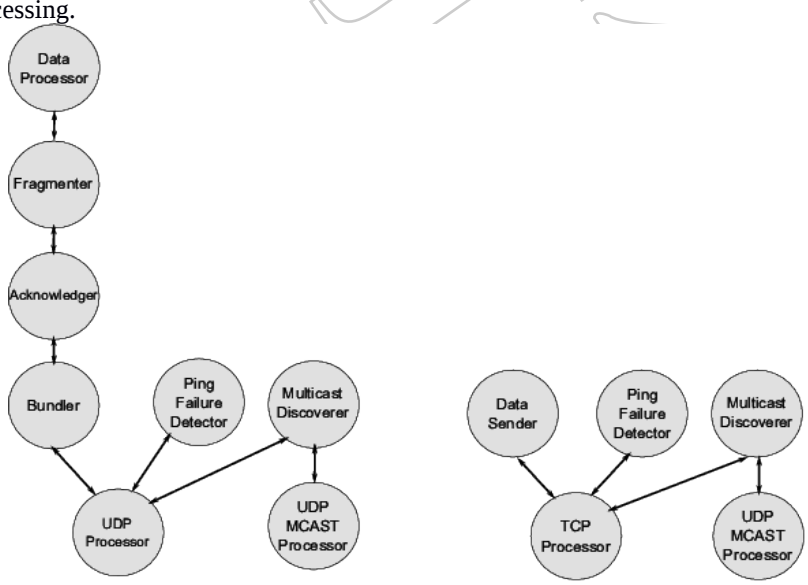


Figure 12. UDP and TCP stacks in jLiME.

Two default stack implementations, depicted in Figure 12, are provided by jLiME: an UDP-based stack and a TCP-based stack. Each stack might be used in different scenarios. TCP is a stream-oriented protocol, used when a reliable channel must be established between two networked processes.

On the other hand, UDP is packet or “datagram”-oriented and does not provide any guarantee that after sending an array of bytes it will arrive to destination. UDP has the advantage of consuming much less network resources than TCP as no connection must be established nor maintained between nodes. In jLiME, however, we built a series of MPs to overcome UDP unreliableness. Firstly, to adjust to the UDP maximum datagram limit, we created a MessageFragmenter that divides the original packet into smaller pieces and assembles them on the receiving side. Secondly, an Acknowledge processor confirms received messages, and throttle sending if a given amount of confirmations are missing. This helps to maintain a steady stream of UDP packets by not overloading the receiver. Thirdly, a Packet Bundler merges small messages in a bigger message or “bundle” to reduce datagram overhead. Naturally, TCP already implements these techniques, but unlike the UDP stack, they are not optional.

Additionally, for both types of stack a custom discovery protocol was designed and implemented in two forms: Multicast Discovery and Ping Discovery. The Multicast variation sends UDP multicast messages to a given multicast address and expects responses from nodes. The Ping-based discovery simply sends UDP messages to a range of addresses and waits for responses. Similarly, a ping-based protocol was created to track the status (alive or defunct) of nodes in the cluster.

Lastly, streaming capabilities might be added to the jLiME stack by implementing the Streamer interface. The TCP-based implementation of this interface is straightforward as it is a stream-oriented protocol. The UDP Message Processor provides streaming capabilities by reassembling packets in the order they were sent.