HAL
archives-ouvertes.fr

A Generic API for Load Balancing in Structured P2P Systems

Maeva Antoine, Laurent Pellegrino, Fabrice Huet, Françoise Baude

▶ **To cite this version:**

Maeva Antoine, Laurent Pellegrino, Fabrice Huet, Françoise Baude. A Generic API for Load Balancing in Structured P2P Systems. 26th International Symposium on Computer Architecture and High Performance Computing, Oct 2014, Paris, France. pp.138 - 143, 10.1109/SBAC-PADW.2014.17 . hal-01101688v2

HAL Id: hal-01101688
https://hal.inria.fr/hal-01101688v2

Submitted on 15 Jul 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Generic API for Load Balancing in Structured P2P Systems

Maeva Antoine, Laurent Pellegrino, Fabrice Huet, Françoise Baude
University of Nice Sophia Antipolis, CNRS, I3S, UMR 7271
06900 Sophia Antipolis, France
firstname.lastname@inria.fr

***Abstract*—Real world datasets are known to be highly skewed, often leading to an important load imbalance issue for distributed systems managing them. To address this issue, there exist almost as many load balancing strategies as there are different systems. When designing a scalable distributed system geared towards handling large amounts of information, it is often not so easy to anticipate which kind of strategy will be the most efficient to maintain adequate performance regarding response time, scalability and reliability at any time. Based on this observation, we describe the methodology behind the building of a generic API to implement and experiment any strategy independently from the rest of the code, prior to a definitive choice for instance. We then show how this API is compatible with famous existing systems and their load balancing scheme. We also present results from our own distributed system which targets the continuous storage of events structured according to the Semantic Web standards, further retrieved by interested parties. As such, our system constitutes a typical example of a Big Data environment.**

***Keywords*—API, Load Balancing, Modularity, Structured P2P**

I. Introduction

With the advent of Big Data, it becomes incredibly difficult to manage realistic datasets on a single machine. To face the incredible amount of information to manage, many solutions have been proposed. One is to use structured Peer-to-Peer (P2P) systems, which are an efficient and scalable solution for data storage and retrieval in large distributed environments. Some NoSQL databases are based on underlying P2P structures, like Cassandra [1] which is fully distributed and uses consistent hashing which results in a Chord-like topology.

However, one key issue with distributed systems concerns load balancing, in particular for systems geared towards data dissemination. A few nodes can quickly become a bottleneck since a biased data distribution can lead to large workloads sent to very few nodes. Indeed, real world datasets are known to be highly skewed [2]. The main reason for this lies in the variation of size, popularity and lexicographic similarities among resources. Information that is stored, shared or more generally manipulated can come from different sources (world wide data expressed in various languages) and be more or less structured using various formats [3] providing reasoning capabilities. Furthermore, imbalances in a distributed system may also be caused by an unfair partitioning of network identifiers, frequent peer arrival and departure or heterogeneity in terms of bandwidth, storage and processing capacity between peers.

To address load imbalance issues in structured P2P networks, especially regarding data distribution, several load balancing strategies have been proposed based on replication or relocation. The model followed by these strategies usually consists in controlling resources and/or peers location. However, many variants are conceivable based on indirection, identifiers, range space reassignation or virtual peers. Moreover, designing a load balancing solution requires to consider parameters such as the overload criteria to take into account, how overload is detected, and how load information is exchanged. This variety of parameters has led to the definition of multiple solutions that often differ by minor but subtle changes. To date, this topic is still relevant [4].

When designing a scalable distributed system, it is often not so easy to anticipate which kind of load balancing strategy will be the most efficient to ensure adequate performance for users and prevent node failure. In this paper, we propose to describe the main concepts behind a generic API for load balancing in structured P2P systems. Our contribution simplifies coding and maintenance by enabling the integration of different strategies in a system, with minimal impact on the existing business code. More precisely, we will focus on the context of data management systems. Indeed, this work [5] was motivated by the building of a distributed platform for data storage and retrieval especially geared towards situational-driven adaptability, taking the form of an event marketplace platform[1]. However, the general ideas presented in this paper can be applied on other types of truly distributed systems. To this aim, we provide a guide of what criteria are important to define and the essential principles to think about before implementing a load balancing strategy. We propose to decompose into components the main features arising from a load balancing mechanism. This enables changing only a part of a strategy without having to impact the other components. Our contribution is to provide a synthetic and operational vision of how different bricks that form a load balancing strategy articulate, and at which time in its life cycle these bricks are used. Therefore, our solution is especially useful when it becomes necessary to experiment with various load balancing strategies in a system, prior to a definitive choice for instance.

The rest of the paper is structured as follows. Section II introduces some existing load balancing solutions we find relevant in our context. Section III describes what we consider as the main elements that make up a load balancing strategy. Section IV presents our common API to implement any kind of strategy. Section V shows how our API can be applied to the previously mentioned systems from the literature. Section VI describes the experiments and presents the results obtained to balance the load of our own distributed storage system. Finally, Section VII concludes the paper.

[1] http://www.play-project.eu/

II. Existing Systems

Many papers propose load balancing solutions for distributed systems, using various strategies. In the following, we focus on three different systems, each implementing its own load balancing strategy. Although the chosen papers do not constitute an exhaustive list of load balancing solutions, they are representative of existing works. Indeed, these strategies are applied on various P2P systems (CAN, Chord) and in different contexts (publish/subscribe, data storage). Load balancing is triggered at different states: when a new peer joins the system (to implement horizontal elasticity for instance), when inserting data, or periodically. Besides, and perhaps more importantly, these papers are among the most-cited for the topic of structured P2P systems. We focus on structured P2P systems as they are an efficient solution to ease scalability. Furthermore, nowadays, many Big Data systems employ P2P as an underlying structure [6].

1) Rao et al.: In [7] the authors suggest three different strategies based on virtual peers to address the issue of load imbalance in P2P systems that provide a Distributed Hash Table (DHT) abstraction. This paper proposes a general solution, not especially dedicated to data load balancing. Each physical node is responsible for one or more virtual servers, whose load is bounded by a predefined threshold. A node is considered as imbalanced depending on this threshold: heavy if its load is above, light otherwise. The proposed solutions are meant to transfer the load between heavy and light nodes by moving virtual peers only. The first scheme involves two peers to decide whether a load transfer should be performed or not. A peer contacts a random peer, and both exchange their load information. If one of them is heavy and the other one is light, then a virtual server transfer is initiated. The second scheme relies on directories indexed on top of the overlay. Each directory, indexed on a node, stores piggybacked load information from light nodes. When a node receives a message from a heavily loaded node, it looks at the light nodes in its directory to transfer the heaviest virtual server from the heavily loaded node to a lightly loaded one. Finally, the third variant matches many heavily loaded nodes to many lightly loaded nodes, still using directories. A node holding a directory receives load information from both heavy and light nodes. This node periodically performs an algorithm to calculate how to balance the load between all these nodes. Solutions specifying which virtual servers should be transferred to which nodes are then sent to the concerned nodes.

2) Gupta et al.: Gupta et al. [8] exploit the characteristics of CAN [9] and their publish/subscribe system (Meghdoot) properties to balance the load when new peers join the system. Each peer periodically propagates its load to its neighbors. When a new peer wants to join the system, it contacts a known peer in the system, responsible for locating the heaviest loaded peer. The authors distinguish subscriptions load from events load. To balance subscriptions matching load, the idea is to split a heavy peer's zone so that its number of subscriptions is evenly divided with the peer that joins. The second solution, to address event propagation load imbalance, creates alternate propagation paths by using replication: when a new peer p_j joins a peer p_i overloaded by events, the zone from p_i is replicated on p_j (including its subscriptions).

3) Byers et al.: In [10] the authors investigate the direct applicability of the power of two choices paradigm [11] on the Chord [12] P2P network for addressing load imbalances in terms of items per peer. A node that wishes to insert an item applies d hash functions on the item key and gets back d identifiers (each hash function is assumed to map items onto a ring identifier). Afterwards, a probing request is sent for each identifier computed previously and the peers managing the identifiers answer with their load. Once load information is retrieved, the peer with the lowest load p_{low} is adopted for indexing the item. The other $d-1$ peers that were contacted but not selected receive a redirection pointer (key space identifier) to p_{low} for the corresponding item. Thus, a lookup can be achieved by using only one hash function among d at random.

III. Load Balancing Differentiators

Although they seem very different, all the load balancing strategies cited above and most other existing solutions rely on the same principle. A peer decides to move a given amount of *load* to a certain *target* which will become responsible for the *load* being moved. The decision to move load always comes after a load comparison with a given *source* of information. It is very common to trigger this load comparison during a specific state of the system such as network construction, data insertion or periodically. Overall, we identified the following differentiators to establish a load balancing strategy. They represent the main concerns to focus on in order to develop a strategy.

a) Criteria: Before fixing load imbalances, disproportion in terms of load must be detected. This implies to know which load criteria are involved and how their variation is measured on peers. This differentiator defines which load variations are considered and to which resource(s) (CPU, bandwidth or disk usage) and operation(s) (e.g. item lookup, item insertion, etc.) they refer.

b) Load State Estimation Algorithm: This step consists in defining whether a peer is experiencing an imbalance or not, and how this decision is made. Usually, a peer relies on a source of load information containing aggregated remote information (see differentiatior g)) or uses local information by comparing its local load(s) with predefined threshold(s).

c) Load Balancing Decision: The decision to trigger load balancing often differs from a load balancing strategy to another. This differentiator aims to identify when the decision to evaluate load state is triggered. Consequently, it is related to the time at which the whole load balancing mechanism is triggered and will necessarily impact how a load balancing implementation is welded to an existing system business code.

d) Load Balancing Mechanism: The mechanism identifies which well known solution is applied to move load from a peer to its *target*. It may consist in using *virtual peers*, *redirection pointers* or even *range space reassignation*. It helps checking whether prerequisite abstractions, required to define a given load balancing strategy, are available or not.

e) Load to Move: Once an imbalance is detected, the next stage is to fix it. This implies to know what is the load to move. This differentiator defines the amount of load to move from a peer to its *target* but also which part.

f) Target: Given an unbalanced peer p, its target is a set of peers used to balance its load with. In other words, it describes who receives the load when load balancing is triggered.

g) Load Information Exchange: A strategy optionally embeds a mechanism to exchange information. It is often used to compare the local load to an average system load

estimated through exchanged information. This differentiator defines when estimations are transferred (if they are), from who and how. Once received on a peer, these estimations compose a *source* of information.

h) Load Information Recipients: Given a peer p, recipients are peers that share load information with p. They are mainly used to build a *source* of information involved in the load balancing decision process.

IV. Generic API for Load Balancing

Defining a generic load balancing API requires to identify key abstractions suitable to model any strategy. In this section, we define the components and functions of our API, based on the differentiators presented above. An approach based on hierarchical components was deliberately used because components enable modularity and cohesion [13], which eases reusability.

A. High-level Abstractions

Features associated to differentiators a) to f) relate to the management of load balancing and could be gathered in a so called *Load Balancing Manager* component. By pushing our analysis deeper, we may argue that differentiators b), c) and e) to f) identify two separate subcomponents. Indeed, the first group of differentiators relates to the detection of imbalances (*Imbalance Detector*), whereas the second (*Load Balancer*) captures the method and the information required to balance the load in case of imbalance. Finally, differentiators g) to h) are merely involved in the process to give feedback about resource utilization per criterion to peers. In a component-oriented approach, this could be modelized as a *Load Information Manager* with a subcomponent, dubbed *Load Information Exchanger*, in charge of exchanging load information.

Figure 1 illustrates these components in charge of isolating load balancing features on each peer. In addition to the components presented above, the figure sketches an additional one, named *Load Information Registry*, that aims to link the two main composite components (*Load Balancing Manager* and *Load Information Manager*), since each may run in its own flow of control.

Components are wired together by calling actions on other components. Some actions carry $Load\ Information$ which contain the following values:

- $peer$: the peer sending its $load\ information$. Can be a peer identifier, a reference, etc.
- $criterion$: type of $load$ (disk space, CPU consumption, bandwidth, etc.).
- $load$: load of the $peer$ for a given $criterion$.

These attributes and their value can be expressed in the form of a key/value list. Optional elements such as $optimal\ load$, $internal\ threshold$ or a $timestamp$ can also be included. Details about internal components actions and their behavior are given in the next subsection.

B. Core API

Function calls defined below capture the core of load balancing strategies, classified per component. The signature for required functions is given in a simple untyped pseudo language, thus allowing any particular implementation.

Before entering into the description of the API for each simple component, it is worth noting that the two main composite components identified previously respectively expose a **perform_one_load_balancing_iteration()** and a **perform_load_information_exchange()** function. They act as entry points for peer instances to execute one step of the two complementary composite components code, thus orchestrating in which order functions introduced below are run.

Load Information Exchanger: This component is responsible for sending the peer's $Load\ Information$ and receiving $Load\ Information$ from other peers in the network.

- **exchange_load_information**($recipients$, $load\_information$) $\rightarrow load\_information$

A peer sends and receives $Load\ Information$ from other peers, for a given load *criterion* (storage, CPU, etc.) and a corresponding amount of *load*. The **exchange_load_information** function may return *Load Information* from pull calls or periodically sent by other peers, that will be directly used by the *Load Balancer* component or stored in the peer's *Load Information Registry* (see details below). A push call is used when a peer wants to unilaterally notify *recipients* (a given number of peers: neighbors, all peers, a random peer, etc.) about its load state *Load Information*.

Load Information Registry: This registry stores all *Load Information* received by a peer. Optionally, time can be taken into account when storing information as it is possible to maintain synchronous clocks using protocols such as NTP.

- **register**($load\_information$)
- **get_load_report**($criterion$, $peers$) $\rightarrow load\_information$

The **register** function writes into the registry *Load Information* received by the peer's *Load Information Exchanger*. The **get_load_report** operation provides *Load Information* for a given set of *peers* according to a certain *criterion*. This estimation is calculated thanks to the *Load Information* messages received and stored earlier. The returned *Load Information* can help estimate the overall average load or the load of a given peer, for example. There can be no result if the calling peer has not recently received any *Load Information* message from the concerned *peer(s)*.

Imbalance Detector: Default behavior is to check if a load criterion is unbalanced (overload or underload), in order to trigger a load balancing strategy.

- **make_decision**($criterion$) $\rightarrow load\_state$

Using a given algorithm, this function determines whether to induce a load balancing strategy or not, according to a given *criterion*. This operation is basically meant to return an enumerated type: *overloaded* or *underloaded* if a rebalance is necessary, *normal* otherwise. The returned value may depend on a threshold value or not, typically to detect overload or underload. If a threshold value is used, it can be calculated using *Load Information* provided by the *Load Information Manager* (locally, from the *Load Information Registry* using **get_load_report**, or remotely by contacting peers with **exchange_load_information**).

Load Balancer: This component is responsible for balancing the load.

- **select_load_to_move**($load\_information\_manager$, $criterion$, $load\_state$) $\rightarrow load\_to\_move$
- **select_target**($load\_information\_manager$, $criterion$, $load\_state$) $\rightarrow target$

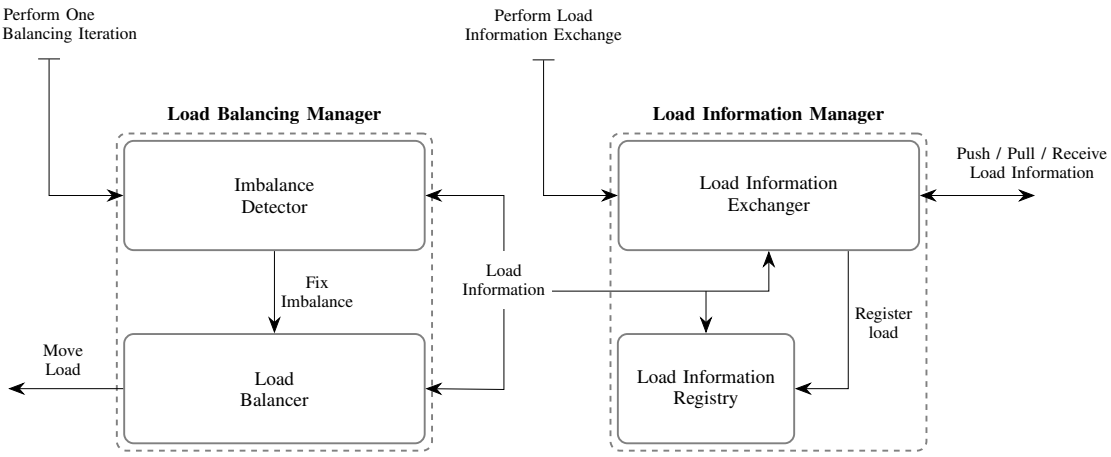

Figure 1: Basic abstractions per peer for a generic API.

- **rebalance**($criterion$, $target$, $load\_to\_move$)

The **select_load_to_move** operation is necessary to calculate the amount of *load to move* from one peer to another. Optionally, it is possible to use local or remote information from the *Load Information Manager* to determine how much load has to be moved. The **select_target** function is responsible for finding which peer(s) will receive this *load to move*. To do so, it is possible to query the *Load Information Manager* but it is not mandatory (*target* can be a random peer, a new peer, etc.). Finally, the **rebalance** method is used to move the *load to move* between the calling peer and the *target*.

V. API Review on Existing Systems

This work was motivated by the building of our own distributed storage system, for which we wanted to apply the most suitable load balancing strategy. Once implemented [5], we picked various relevant papers from the literature (introduced in section II) to see if they could validate our generic API, too. Table I presents how these strategies, although very different at first sight, match our differentiators. Next, we describe how our API could prove successful in implementing these strategies.

Rao et al.: Peers periodically push their *load information* (*Load Information Exchanger*) to a set of nodes maintaining a directory (*Load Information Registry*). *Load information* contains the load of each virtual server of a peer and the peer's internal *threshold*. Each peer p periodically compares its load $load_p$ for a given load criteria to its $threshold_p$ (*Imbalance Detector*). Depending on the peer's load state, the paper proposes three rebalancing strategies (*Load Balancer*):

1) If $load_p < threshold_p$, p is underloaded and triggers a rebalancing. A random node is picked (**select_target**) and its load sent to p (*Pull Load Information* via *Load Information Exchanger*). If the random node is heavy, then a virtual server transfer (**select_load_to_move**) may take place between the two nodes (**rebalance**).
2) If $load_p > threshold_p$, p is overloaded and contacts one of the peers holding a directory to request a light peer (**select_target**) and which virtual server (**select_load_to_move**) should be moved (*Pull Load Information* via *Load Information Exchanger*). Afterwards, **rebalance** is called.
3) If $load_p > threshold_p$, p can also send its *load information* to a peer dir holding a directory (*Push Load Information* via *Load Information Exchanger*). After dir has received enough information from heavy and light nodes, dir performs an algorithm to pick which virtual server p should send (**select_load_to_move**) to which light node (**select_target**). The solution is then sent back to p, to start the **rebalance** process.

The workflow associated to the second strategy is depicted in Figure 2. Steps are numbered to sketch the sequence of actions involved in a typical load balancing iteration with three peers. Arrows between function calls depict remote communications.

Gupta et al.: Peers periodically exchange their load information with their neighbors (*Load Information Exchanger*), as well as an estimated list containing the most heavily loaded peers they know (from their *Load Information Registry*). Load balancing is only triggered (*Imbalance Detector*) when a new peer p wants to join the system. The first step of the rebalancing process (*Load Balancer*) is for p to find an overloaded peer in the system (**select_target**). To do so, p sends a $pull\ request$ to a random peer. Then, the random peer will look at its registry in order to tell p which node ($target$) is the most overloaded to its knowledge. Finally, $target$ is contacted by p. If the overload is due to the amount of subscriptions, p will split its zone with $target$ and receive half of $target$'s subscriptions. Otherwise, if $target$ is overloaded because of its processing load due to event propagation, p will replicate $target$'s zone and subscriptions (**select_load_to_move** and **rebalance**).

Byers et al.: A peer having to insert a data item into the system triggers the process (*Imbalance Detector*). This peer applies n hash functions on this item. Then, it contacts each peer associated to an hash function (*Load Information Exchanger*) to pull load information concerning the amount of items already stored by each of these peers. The *Load Balancer* then selects the lightest peer (**select_target**) and sends it the item to be inserted (**rebalance**).

VI. Generic API Evaluation on EventCloud

This work was originally motivated by the PLAY project[2] which is a platform that allows for "event-driven interaction in large highly distributed and heterogeneous service systems". In this context, we have developed the EventCloud (EC) [14] middleware, a structured P2P system allowing continous injection of big amounts of events and their retrieval.

[2]http://www.play-project.eu

	Rao et al.	Gupta et al.	Byers et al.
Criteria	Resource agnostic (storage, bandwidth or CPU) but only one	Subscriptions and data popularity	Number of data items per peer
Load State Estimation Algorithm	Given L_i the load of node i (sum of the loads of all virtual servers of node i) and T_i a target load chosen beforehand, a node is heavily loaded if $L_i > T_i$, lightly loaded otherwise	Always triggering rebalancing when a new peer joins the system	Always triggering rebalancing when receiving an item to insert
Load Balancing Decision	Periodically, on each peer	When a peer joins the system	Upon the insertion of an item (data) on the entity that performs the insertion
Load Balancing Method	Virtual peers transfer (with no virtual peer split or merge)	Range space reassignation or replication (zone + subscriptions)	Power of two choices paradigm
Load to Move	One of the overloaded peer's virtual server	Half of a heavy peer's subscriptions or replication of a heavily loaded peer	The item to be inserted
Target	A random peer, an underloaded peer or the best underloaded peer according to the scheme used	The heaviest peer in the system known by the new peer joining the system	The least loaded peer among those contacted in source for a given item
Load Information Exchange	Random probing for the first scheme (*pull*). Periodic load advertisement from lightly loaded peers (*push*) and sampling from heavily loaded peers (*pull*) with the second scheme. Third scheme implies load exchange from a peer to a directory (*push*)	Periodically, peers update neighbors about their load (*push*) and share their list that contains the k most heavily loaded peers detected	The peer that wants to insert an item computes d hash values and contacts the associated peers to retrieve their load (*pull*)
Load Information Recipients	A peer managing a random id for the first scheme. Directory associated to some peers for the second and third scheme	One hop neighbors	For d hash functions applied on an item to insert, the n peers managing the computed hash values

Table I: Load balancing strategies mapped to differentiators.

A. EventCloud

The EC is a Java software block that offers the possibility for services to communicate in a loosely coupled fashion thanks to the publish/subscribe paradigm but also to store and to retrieve past events in a synchronous manner. Events are semantically described as a list of quadruples. Quadruples are in the form of ($graph$, $subject$, $predicate$, $object$) tuples where each element is named an RDF term in the RDF [3] terminology.

Load balancing with the EC was motivated by the fact that some RDF terms are more popular than others (especially *predicates*) but also because many RDF terms share common prefixes since they are URIs. In that case, one or a few adjacent peers from the identifier space manage most data while many index no information.

B. Load Balancing Strategies

The focus is on two key aspects with the EC: load criteria and load information exchange. The former is to provide a load balancing method that allows to consider the unbalance of multiple different criteria. The latter aspect concerns information being exchanged. The more information is spread, the more precise the average system load estimate is. To assess the API, two load dissemination strategies are proposed, namely *absolute* and *relative*. The first aims to detect imbalances without exchanging information between peers. Load balancing is triggered when a local threshold is exceeded. In contrast, the second relies on exchanged information to detect whether an imbalance is experienced.

C. Results

Relative and *Absolute* strategies have been implemented and assessed with micro benchmarks using real data extracted from a Twitter data flow and up to 32 peers deployed on the French Grid'5000 testbed [15]. The workload is about 10^5 quadruples.

Before evaluating both strategies, an experiment has been performed to see what could be the best distribution. The scheme consists in injecting the workload on a single peer and once all quadruples have been stored to start load balancing iterations. Each load balancing iteration consists in picking a new peer from a preallocated pool of peers to make it join the most loaded one in the network, thus simulating an oracle. The action is repeated until having a network containing 32 peers. The number of peers is deliberately low as we aim to assess the API in this paper, not the load balancing strategies. To compare results for a same configuration (i.e. same workload and number of peers), a good estimator is the coefficient of variation, also known as the relative standard deviation. It is expressed as a percentage by dividing the standard deviation by the mean times 100. In the following, we use this estimator to compare strategies. For information, the coefficient for the best possible distribution in this configuration and for this dataset is 69.5% by using the oracle assumption.

Then, we ran experiments for the *absolute* and *relative* schemes. For the absolute one, the threshold value is set to the number of quadruples divided by the final number of peers. The relative strategy does not rely on global knowledge but triggers load balancing when local measurements on peers are greater than or equal to 1.1 times the estimate value computed by receiving load information from immediate neighbors. The last value was set according to emperical evaluations that let suppose the best distribution is achieved for this value. The *absolute* strategy achieves 119.75% while the *relative* 96.57%.

When correlated with the results obtained for the first experiment that exhibit the best distribution (69.5%) due to the "oracle" assumption, the relative standard deviation is almost twice as large (119.75%). Similarly, the relative strategy performs worse than the oracle one but achieves a

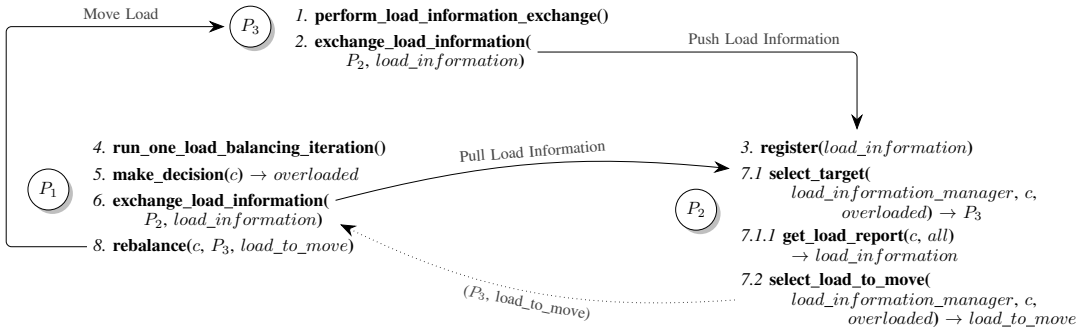

Figure 2: Workflow associated to the second scheme proposed by *Rao et al.*

better distribution (96.57%) than the absolute strategy and this without using global knowledge. Besides, since more peers are receiving RDF data, more nodes are involved to answer subscriptions with the pub/sub layer, thus increasing the throughput in terms of notifications received per second for end-users of EC.

Although the analysis of the results is not the central point of this paper, it shows that investigating different implementations for the functions identified in section IV may have strong impact on results. Thanks to the proposed API, the behaviour of the different load balancing stages can be simply changed by writing a new method with less than 10 lines of code in our case. The main reason is that key features of the load balancing workflow are clearly identified. Obviously, the example shown in this section still requires one line of code change and a full code recompilation to switch from a component implementation to another. In our case, an alternative based on dynamic class loading could be used [16]. In this situation, some code redeployment is required. Moreover, synchronization between nodes have to be taken into consideration to prevent inconsistent states due to stale information that could transit during the transition from a component implementation to another on peers.

VII. Conclusion

In this paper, we have described concepts behind the building of an API for load balancing in structured P2P systems. We have presented three different schemes from some of the most-cited papers for the topic. Strategies are triggered at various moments, impact more or less peers and require to move or replicate data. By decomposing a strategy into essential differentiators, we have shown it is possible to implement these different solutions using our generic API. Regarding the programming aspect, the API allows to separate the code concerning load balancing from the rest of the system. To further assess its utility, the API has been used to evaluate different load balancing methods on our own distributed system [5]. Although we presented this API in the context of structured P2P systems, the ideas introduced in this paper could be adapted to many Big Data systems using an underlying P2P structure. Finally, two interesting perspectives are possible. The first is about the flexibility of the proposed API which can lead to implement adaptive load balancing strategies, for example, by changing one of the key features in a load balancing workflow at run-time [17]. The second is related to the properties of our system presented in VI. Although it provides scale-up load balancing only, since events are stored in a sustainable manner, supporting scale-down load balancing by extending our solution would enable better resource usage through (even autonomic [18]) elasticity.

References

[1] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[2] S. Kotoulas *et al.*, "Mind the data skew: distributed inferencing by speeddating in elastic regions," in *Proceedings of the International conference on World Wide Web*. ACM, 2010, pp. 531–540.

[3] O. Lassila and R. R. Swick, "Resource Description Framework (RDF) model and syntax specification," 1999.

[4] P. Felber, P. Kropf, E. Schiller, and S. Serbu, "Survey on load balancing in peer-to-peer distributed hash tables," 2014.

[5] M. Antoine, L. Pellegrino *et al.*, "Towards a generic API for data load balancing in structured P2P systems," INRIA, Research Report 8564, 2014. [Online]. Available: http://hal.inria.fr/hal-01022722

[6] W. X. Goh and K.-L. Tan, "Elastic mapreduce execution." in *CCGRID*, 2014, pp. 216–225.

[7] A. Rao *et al.*, "Load balancing in structured P2P systems," in *Peer-to-Peer Systems II*. Springer, 2003, pp. 68–79.

[8] A. Gupta *et al.*, "Meghdoot: content-based publish/subscribe over P2P networks," in *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag, 2004, pp. 254–273.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network*. ACM, 2001, vol. 31, no. 4.

[10] J. Byers *et al.*, "Simple load balancing for distributed hash tables," in *Peer-to-peer Systems II*. Springer, 2003, pp. 80–87.

[11] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[12] I. Stoica, R. Morris *et al.*, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[13] G. T. Heineman and W. T. Councill, "Component-based software engineering," *Putting the Pieces Together, Addison-Westley*, 2001.

[14] L. Pellegrino, "Pushing dynamic and ubiquitous event-based interaction in the Internet of services: a middleware for event clouds," PhD Thesis, University of Nice Sophia Antipolis, Apr. 2014.

[15] F. Cappello, E. Caron, M. Dayde *et al.*, "Grid'5000: A large scale and highly reconfigurable grid experimental testbed," in *Proceedings of the IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society, 2005, pp. 99–106.

[16] S. Liang and G. Bracha, "Dynamic class loading in the Java virtual machine," *ACM SIGPLAN Notices*, vol. 33, no. 10, pp. 36–44, 1998.

[17] N. DePalma, K. Popov *et al.*, "Tools for architecture based autonomic systems," in *International Conference on Autonomic and Autonomous Systems (ICAS)*. IEEE, 2009, pp. 313–320.

[18] A. Al-Shishtawy and V. Vlassov, "Elastman: autonomic elasticity manager for cloud-based key-value stores," in *Proceedings of the international symposium on High-performance Parallel and Distributed Computing*. ACM, 2013, pp. 115–116.