Deploying a High-Performance Context-Aware Peer Classification Engine

Mircea Bardac, George Milescu, and Adina Magda Florea
Automatic Control and Computers Faculty, Computer Science Department
University POLITEHNICA of Bucharest, 313 Splaiul Independentei, Bucharest, Romania
Email: {mircea.bardac, george.milescu, adina.florea}@cs.pub.ro

***Abstract*—This research presents a generic approach for context-aware entity classification with emphasis on integration and use of contextual information in Peer-to-Peer systems. The designed peer classification engine isolates high-latency update processes in order to minimize the latencies of the lookup queries. By using a key-value data-store with support for sorted sets, high complexity context-classifying functions can be executed asynchronously without impacting the lookup queries. The performance of the system is evaluated through experimental and complexity analysis, identifying directions for improving and scaling the peer classification engine. As ubiquitous computing evolves and becomes part of everyday life, the designed context-aware classification engine provides a basis for deploying the next-generation network-based services.**

***Keywords*-classification; *context-awareness*; *peer-to-peer*; *BitTorrent*.**

I. Introduction

Contextual information is becoming more important as ubiquitous computing evolves and integrates into everyday life. An increasing need for context-aware network services has led the research community to investigate means of managing contextual information within the scope of existing network protocols. Whether it is data location, data availability or some other characteristic of the data, the context can play an important part in defining the interaction model and how data is being accessed.

Rapid content distribution and various optimizations have boosted the use of Peer-to-Peer (P2P) solutions. Scientific research has focused on improving existing P2P protocols for delivering better performance and providing higher availability for various network topologies, overlays and swarms.

The Peer Classification Engine presented in this paper provides a generic approach for context-aware classification for any type of entity or data inside a network. Entities can be peers in Peer-to-Peer networks, sensors in Wireless Sensor Networks, computer nodes in a cluster, or any other networked entities. The evaluation of this solution has been performed considering the Peer-to-Peer paradigm, using BitTorrent interactions for contextual evaluation. BitTorrent was chosen as it is the most widespread Peer-to-Peer protocol implementation currently in use, accounting for most of the Internet traffic [1].

The solution is designed to ensure fast responses for *lookup queries*, while asynchronously processing the slow *update queries* in the background.

A. Context-Awareness in P2P systems

In a world where providing differentiated services based on various conditions is becoming important both from the technical and economical points of view, context awareness plays an integral part of designing the next-generation network-based services. Various models such as [2] have been developed to accommodate the needs for contextualized data retrieval in P2P networks. Even so, contextualized information is still difficult to include into testing and deployment platforms such as the ones described in [3] and [4], and has not been yet considered for performance evaluation [5].

Monitoring platforms such as [6] and [7] are the best suited for extracting contextual information but, even so, this information must be used by the service-providing layers.

Existing P2P context-aware applications usually rely on location data [8] as the most easy to determine context. Search also benefits from contextual information [9]. P2P applications running on mobile devices also take into account device capabilities in order to facilitate an adapted service discovery mechanism for the peers [10], or for processing mobile data over large mobile network environments [11].

A system that simplifies context-aware classification is needed for supporting the increased demand of contextualized services. The system must scale to a large number of contexts without affecting the performance of the consumer applications within the P2P network. These are the basic requirements that led to the development of the proposed solution.

B. Storage and performance constraints

One of the problems in classifying peers in a large-scale network with numerous contexts is the underlying data structure used for storing the classification information. Regular databases have been considered as they are commonly used for storing and retrieving information easily. Unfortunately, as the entire system aims for high performance lookup queries and the classification information has a fixed format, the complexity of a relational-database was considered an impediment and other solutions were evaluated.

Structured storage solutions with simple key-value mappings, modeled after the NoSQL paradigm [12], are considered the best choices in terms of design complexity and provided performance. Therefore, the storage does not have fixed table schemas and most operations scale horizontally, making this

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-133-5

suitable for describing large numbers of contexts. An in-memory storage solution brings even a higher read/write throughput. If needed, this solution can scale beyond a single machine.

The structured storage is the core part of the Peer Classification Engine. The peers in the P2P network are the ones providing updates for the Peer Classification Engine, and they are also the ones benefiting from the classified results. Depending on the desired functionalities and supported contexts, updates are being processed by the system, and peers, content and other information are classified into several classes, as described in Section II.

In order to provide a high-performance peer classification solution, the major design goal of the system is to *minimize latencies for lookup queries* while doing most peer classification computations during the updates. Previous designs [13] have shown that the computationally intensive processes should be isolated, and common operations, in this scenario - lookup queries, should be optimized.

This paper is structured as follows: Section I presents an overview of P2P, context-awareness in P2P and the storage and performance requirements that needed to be solved, Section II details the design decisions taken for deploying the Peer Classification Engine, Section III presents a complexity analysis of the underlying operations implemented in the proposed solution. Section IV describes the experimental analysis of the Peer Classification Engine, focusing on system latency, resource consumption and presenting some scalability issues. Section V makes an overview of the planned future work as identified after the initial deployment of the Peer Classification Engine, and Section VI presents the conclusions.

II. Peer Classification Engine Design

This section presents the design decisions that led to the development and analysis of the Peer Classification Engine. Based on the requirements detailed in Section I, the design covers the impact of the type of queries being handled, the software components of the system and the classification and ranking algorithms.

A. Queries

Depending on whether or not a query changes the peer classification within the system, two types of queries have been identified: update and lookup queries. The performance requirements of the system have been elaborated based on these two operations and their underlying effects.

In a BitTorrent-based P2P system as the one being considered in evaluating the Peer Classification Engine, update queries would be similar to updates received from the peers by the tracker entities in the BitTorrent network. Lookup queries would be similar to the scrapes performed by the peers for gathering information about the swarms. These scrapes are used to retrieve the lists of active peers, but, in a more complex system, a monitoring entity could use the lookup queries to retrieve much more context-rich information, such as the content availability information within a certain geographical region.

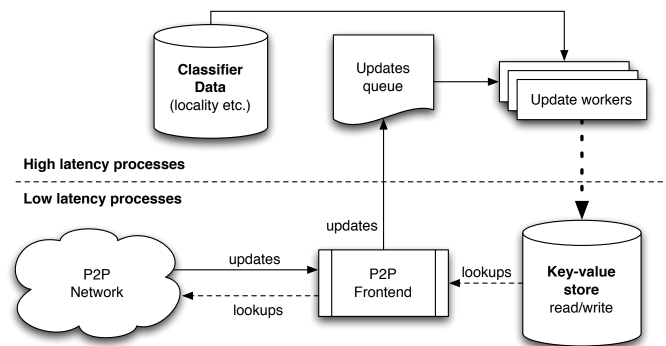

Fig. 1. Peer Classification Engine Design

1) Updates: An *update* coming from a peer pushes a new state to the Peer Classification Engine. The new state may contain various pieces of information about the peer such as its availability (whether it has become online or offline) or content availability (which content is provided by the peer, what parts of the content are provided, etc.). In BitTorrent, *announce messages* are being used by the peers to push updated state information such as number of uploaded/downloaded/corrupt bytes, whether a download has been completed or not, etc.

In the Peer Classification Engine, the new state is used to update the classes the peer belongs to, a high-latency operation by design, given the requirements of the system.

2) Lookups: Queries that retrieve information from the Peer Classification Engine are considered *lookups*. The engine is designed to provide low-latency lookup replies by having the peer classification pre-computed during the updates. In BitTorrent, the *scrape messages* that are used to retrieve the list of peers providing a specific content only contain the hash of the data (torrent) being downloaded.

In the proposed solution, lookup queries can also aggregate information across multiple classes using different weights in order to provide a multi-contextual response for a peer, as detailed in Section III - Complexity Analysis.

B. Components

As detailed in Figure 1, the classification engine relies on the existence of an in-memory *key-value data-store* for managing the peer classes. As previously presented in Section I, a relational database does not meet the performance needs required for high-performance lookup operations. Cassandra [14] has been initially considered as a structured data solution but its complexity added too much overhead to the system. In the end, *Redis* [15] was chosen as a storage solution.

Update queries are being received directly from the peers by the *P2P frontend*, and pushed for later processing to an *updates queue*. This allows asynchronous processing of the updates by one or more *update workers*, which consume the updates from the updates queue. This also implies that, once an update arrives, its effect might not be initially visible in the lookup results. Depending on the number of update workers and on the complexity of the update operations, the time required for the update to reach the key-value store may vary. Lookup queries

Copyright (c) IARIA, 2011.

are also received through the *P2P frontend*, but results are retrieved directly from the key-value data-store.

Figure 1 also presents the separation between the low-latency processes and the high-latency processes. This separation is given by the design requirements as detailed in Section I. Most classifying processes are considered high-latency and therefore require the use of a queue, while the lookup queries are low-latency and should not suffer any performance penalty from other computations except for the data-store reads.

C. Classes and Ranking

The entire functionality of the Peer Classification Engine relies on identifying classes associated with various information coming from the peers inside the P2P network. A class is associated with a context (such as being in a certain geographical region). This association between information and a certain class is done using special functions, called class association functions (CAFs). A class association function ($f_{context_type}(data_packet)$) characterizes only *one type of context* (such as locality, or data availability) – for multiple types of contexts, multiple functions are being used.

In order to provide the best-suited replies for specific contexts, a class association function can also return a rank of a specific piece of information within a class. A class association function will therefore return:

- a *class*, which characterizes the information (for example: the geographical region of a peer);
- an *ID* to identify the information being classified (for example: the peer ID);
- a *rank* of the information in the class (for example: the uplink bandwidth of the peer).

The returned *ID* and *rank* are optional, as there are situations where ranking is not necessary, such as for lookup queries. Classification can be used both for the update and for the lookup queries, but with different purposes:

- *update queries* contain information on various changes in a peer's state. These changes affect the peer's classification within several classes, and several CAFs can be used. Classification within a class may require ranking, depending on the context described by the class;
- *lookup queries* may be classified in order to identify which context is addressed within the query (for example: which region a query is related to or which data is being looked for).

As the design of the proposed solution requires providing a low-latency lookup response, the class association functions used on lookup queries must be fast and take preferably constant time, extracting the class information directly from the lookup query without using external data sources.

Table I presents an overview of the class association functions and their return values depending on the type of a query being processed.

TABLE I
CLASS ASSOCIATION FUNCTIONS FORMS

Query type	Class Association Function form
update	$f_{context}(update_message) = (class, ID, rank)$
lookup	$f_{context}(lookup_message) = (class)$

Typical applications for using classes include:

1) selecting which peers are the most suited for serving a requesting peer based on the context of the requesting peer and the ranking of the other peers for that specific context;
2) selecting which peers are the most suited for providing data to other peers;
3) determining if peer queries should be directed to other storage shards, as described in Section IV, Subsection Scalability Analysis.

Several class association functions as seen in Table II have been defined in order to evaluate the proposed solution for BitTorrent P2P systems. They have been listed based on the types of queries being processed.

TABLE II
CLASS ASSOCIATION FUNCTIONS FOR BITTORRENT P2P QUERIES

Update Queries
$locality_update(update) = (locality_class, peer_ID, rank)$
$availability(update) = (t_availability_class, peer_ID, rank)$
Lookup Queries
$locality_lookup(lookup) = (locality_class)$

A $locality_class$ can be specific to a certain geographic region (one class per region), and multiple locality functions can be used, each one considering a different region size for example - the more functions are used, the longer updating the classes will take. Dynamically choosing the number of classes and size of the classes is a future goal, as described in Section V. The two $locality$ functions differ by their output: the $locality_lookup$ function does not have to compute a rank; it only has to return the class of the peer issuing the query, so that peers could be examined within that certain class.

The $t_availability_class$ (torrent availability class) is torrent specific and contains all the peers that share the specific content identified by the torrent. The $rank$ is given by the completion status (how much of the torrent has been downloaded by $peer_ID$, in percent).

In order to minimize the latency for lookup replies, the results of the $locality_lookup$ function applied on the lookup queries can be cached, thus reducing even more the overhead of the functions and relying only on the deterministic behavior of the data-store operations.

III. COMPLEXITY ANALYSIS

The maximum performance of the provided solution is lower bounded by the theoretical limit of the underlying components, mainly the key-value store and the update workers.

The Peer Classification Engine relies on the use of *sorted sets* in Redis for storing information in classes together with the ranking. One sorted set is used for each class. Using normal

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-133-5

sets is also an option, providing a better complexity but no ranking: adding an item to a normal set is $O(1)$, retrieving the members of a set is $O(n)$, intersecting sets is $O(n \cdot m)$.

The theoretical complexity boundaries achievable using the proposed underlying storage solution are presented below. They are grouped by the type of operations being performed, and they take into account the operations needed for accessing the data structures in the key-value store.

A. Update operations

As mentioned in Section I, most computations should be done on updates in order to minimize the lookup times. Classes are meant to be populated after computing the ranking of peers and the class for each peer. Adding an entry to a sorted set is being done with the ZADD Redis command.

In a BitTorrent P2P network, given the class association functions above, processing an update would require the following operations:

1) identifying the $locality_class$ of a peer - very costly in terms of complexity and time consumed, it requires querying a MySQL database with GeoIP information
2) ranking the peer for the identified $locality_class$; if the rank is considered to be upload bandwidth and the peer reports it, this can be ignored as complexity
3) adding the peer to the $locality_class$ with the given $rank$ in the key-value store (in the sorted set associated with the $locality_class$): $O(log(n))$ complexity, where n is the size of the set.
4) calculating the $rank$ for the peer in the torrent availability class can be ignored as complexity, if it is calculated from the values received from the peer
5) adding the peer to the $t_availability_class$ with the previously calculated $rank$ in the key-value store (in the sorted set associated with the $t_availability_class$): $O(log(n))$, where n is the size of the set.

B. Lookup operations

Lookup queries can be classified as presented in Table II. This means that, for a given lookup query, the location of the peer must be identified. This can be a costly operation, and, in order to provide a low-latency lookup reply as intended in the design, a default class can be assigned on the first query, and the resulting value of the class association function can be cached to be used on subsequent lookup queries.

Lookup replies can be formed by taking into account one context or multiple contexts. This is equivalent to verifying one or more classes in the Peer Classification Engine, which results in retrieving data from one sorted set, or by intersecting multiple sets. The computation becomes more complex as the number of sets being used increases. This might look expensive on the first lookup, but sequential lookups in the same classes can be fed from a cache with $O(1)$ lookup complexity, if nothing changes in those classes.

Lookup into one single class using the ZREVRANGE Redis command for retrieving the best m items (in a sorted set with n items) is $O(log(n)) + O(m)$ complexity.

Intersecting multiple classes using the ZINTERSTORE Redis command is $O(n \cdot k) + O(m \cdot log(m))$ complexity, with n being size of the smallest input sorted set, k being the number of input sorted sets, and m being the number of elements in the resulting sorted set.

IV. EXPERIMENTAL ANALYSIS

A. Peer Classification Engine Deployment

The implementation of the context classes with ranks using sorted sets is partially inspired by the data-structures supported by Redis. Redis is not a simple key-value store but also has support for storing/updating/deleting lists, sets, sorted sets, hashes with atomic operations providing low complexity. Atomic operations and support for transactions also allow keeping data consistent between updates and lookups.

Redis version 2.2.0-rc3 has been compiled and installed on Core(TM)2 Quad Q9550 @ 2.83GHz system with 1GB of reserved RAM in order to perform an experimental evaluation of the Peer Classification Engine. The Redis store was filled with torrent information as received from a tracker and passed through the Peer Classification Engine. Measurements were taken for determining the slow and fast paths within the system, ensuring that the design requirements are being met.

B. Experimental Results

The experiments concentrated on evaluating the following performance metrics in order to determine the impact of the solution on both the system and on the context-aware applications being deployed using the system:

- *update query times*, considering the impact of the location classifying functions;
- *lookup query times*, for single and multi class lookups;
- *memory usage*, as it can bring performance limitations with the extensive use of contexts or in systems with many participating peers.

The measurements were realized considering systems with 256, 512, 768 and 1024 active torrents, and 25 locality contexts. As there is one *torrent availability class* for each torrent and one *locality class* for each locality context, the experiments tested the use of the system for $x + 25$ classes, where x is the number of torrents in the system.

As determined from the complexity analysis, most operations are impacted by the number of peers inside a class at a certain time. Therefore, the experiments tested the system with 128, 256, 512, 1024 and 2048 peers for each torrent. Figure 2 shows that a peak memory use of about 500 MB of RAM is reached for the 1024 torrents with 2048 peers each.

Figure 3 shows the time required for an update to be performed in the system, taking into account the time spent determining the location (identifying the locality context) using a GeoIP database. The average time spent for retrieving the geolocation information is 13.75 msec, making the time spent writing the update to the key-value store insignificant. Under these conditions, the times of an update query do not vary much depending on the number of torrents or peers for a torrent, being 13.88 msec on average.

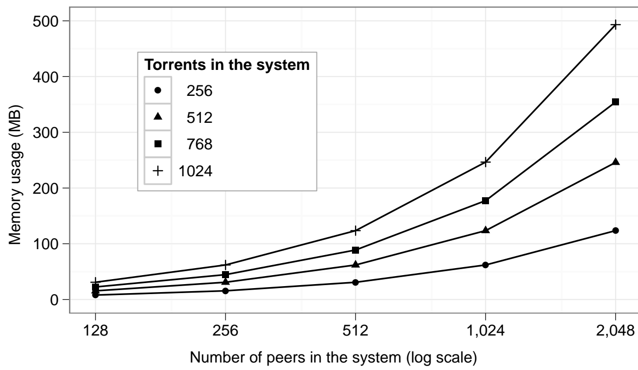

Fig. 2. Memory usage of the key-value data-store

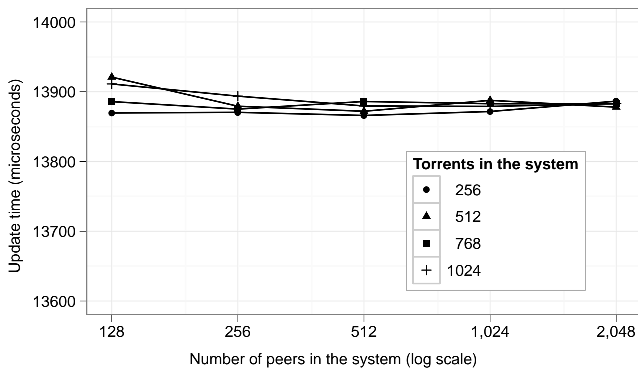

Fig. 3. Update query time

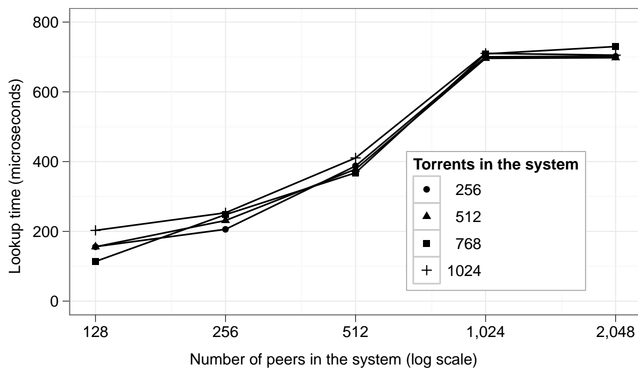

Fig. 4. Lookup query time for one single class

The complexity analysis shows that lookup times are directly impacted by the size of the classes where information is being looked for. As lookup operations are more frequent and results are most likely to be cached, the experimental analysis concentrated on measuring the lookup times only for fetching updated information from the key-value store. Figure 4 shows the lookup times using one single class (checking one single context). As expected from the complexity analysis, this shows that the lookup time increases as the number of peers inside a class grows. Multiple experiment runs have shown that lookup times for the 1024 and 2048 peers are similar.

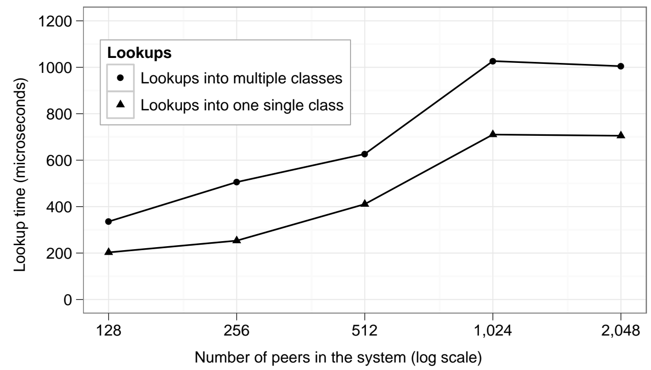

Fig. 5. Lookup query time in the 1024 torrents tests

Redis has internal data-size thresholds used for triggering data reorganization, and the elevated lookup times in the 1024-peers experiments might appear after reaching these thresholds.

Multi-contextual lookups require looking for data into multiple classes, performing intersections between the sorted sets at the level of the key-value store. The time spent looking for data into multiple classes presented a similar trend, although with an added overhead. Figure 5 shows the difference between lookup queries performed into one class versus lookup queries performed into multiple classes (for the 1024 torrent tests). In the experiment, the locality and the torrent availability class were considered. As noticed, intersecting multiple classes adds a significant overhead to the lookup queries, but the impact grows slower as the overhead is limited by the number of results retrieved from the intersection and the size of the smallest input sorted set (which, in this situation, was the locality class set with 25 entries).

The lookup query times ranged from 113 μsec to 1138 μsec, while the update query times averaged at 13883 μsec.

C. Scalability analysis

1) Data sharding: Depending on the content being monitored, number of peers, number of contexts (classes) and other information that can be stored in the in-memory key-value data-store, the Peer Classification Engine might run out of physical memory, and using swap is not an option for a low-latency service. Data sharding can overcome this problem, at the expense of using more machines.

A Redis-based key-value data-storage requires extra middleware logic to implement sharding. Depending on the functions describing the contexts, various data-clustering algorithms [16] can be applied to identify which data needs to be placed inside the same shard. Once the clusters have been identified, data sharding can be used as a solution for dividing data across different machines or even different locations.

In a BitTorrent-based P2P network, peers would make the most use of having data sharded by:

- *torrent hash* - this is the easiest and the most practical method, as content is identified using a hash, and most lookups for information are performed on specific torrent hashes;

- *locality information* - this might bring lower latencies for the lookup replies for local-peers, but brings higher complexity for creating replies for the queries coming from peers which are non-local.

2) Data replication: Most queries on the Peer Classification Engine are expected to be lookups. An increased number of lookup requests or updates in the P2P swarm might increase load on the system, and therefore the latency on the lookup-reply path might become unacceptable. In order to reduce this latency, as reads are very easy to scale, multiple read-only replica storages and/or P2P front-ends can be deployed.

V. Future work

The designed system takes into account that class association functions return a predefined set of classes. Therefore, the number of contexts available in the system can be easily predicted. In continuously evolving systems with a variable number of peers, the use of a fixed number of contexts might lead to uneven resource allocation. A system with 20 peers can benefit more by using a locality association function returning 2 contexts (in this case geographical regions), compared to a 2000-peers system where using 2 locality contexts might lead to an inefficient use of the locality information.

With the given system architecture, *on-the-fly peer-reclassification* can be implemented as an asynchronous high-latency process. The lookup queries will not be affected by an on-going peer reclassification, and, on completion, the new classification might be put in effect. The same behavior can be implemented for sharding. When the load on the system goes over a predefined threshold, an *on-the-fly class-resharing* process can be executed in the background to (1) determine which are the classes (contexts) that can be moved to other data shards, and (2) perform the changes in the data-store.

The key-value store also has support for expiring entries, making it easy to have *contexts that automatically expire* if not updated. This allows outdated information to be automatically removed. Other cleanup operations can be implemented as high-latency processes, without impacting the lookup queries.

VI. Conclusions

The research presented in this paper focuses on designing, implementing and evaluating a Peer Classification Engine that allows contextual information to be used by the peers in a P2P network. The solution provides a flexible approach for defining contexts through class association functions that also support ranking the entities within classes.

The design takes into account that the update and lookup queries impact differently the performance of the entire system. The solution minimizes lookup latencies by offloading the high-latency computations to the update-processing phase. This isolation ensures a predictable behavior of the classification engine as it is being used by the entities in the network.

The solution is not only designed for high-performance lookup queries, but can also provide *high-availability* contextual services through data replication and data sharding. These developments have been explored during the experiments, and, in order to provide an adaptive contextual scaling of the system, dynamic context-based content resharding and on-the-fly peer reclassification are being considered as future work.

Moreover, even though the contexts chosen for evaluating the Peer Classification Engine are Peer-to-Peer specific, the solution can be easily deployed in multiple other environments such as Wireless Sensor Networks, computer nodes in a cluster, or any other networked entities. The proposed solution simplifies the integration and use of contextual information and provides a basis for deploying the next-generation network-based services.

Acknowledgment

This research was supported by CNCSIS - UEFISCSU, project number PNII - IDEI 1315/2008, and by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of Labour, Family and Social Protection through the Financial Agreement POSDRU/6/1.5/S/19.

References

[1] ipoque GmbH, "The Impact of P2P File Sharing, Voice over IP, Instant Messaging, One-Click Hosting and Media Streaming on the Internet," accessed in March 2011. [Online]. Available: http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009

[2] M. Bardac, G. Milescu, and R. Rughinis, "A Distributed File System Model for Shared, Dynamic and Aggregated Data," *17th International Conference on Control Systems and Computer Science (CSCS17)*, vol. 1, pp. 45–51, 2009.

[3] R. Deaconescu, G. Milescu, B. Aurelian, R. Rughinis, and N. Tapus, "A Virtualized Infrastructure for Automated BitTorrent Performance Testing and Evaluation," *Internation Journal on Advances in Systems and Measurements*, vol. 2, pp. 236–247, 2009.

[4] M. Bardac, R. Deaconescu, and A. M. Florea, "Scaling Peer-to-Peer Testing using Linux Containers," in *Proceedings of the 9th RoEduNet IEEE International Conference*, 2010, pp. 287–292.

[5] G. Milescu, M. Bardac, and N. Tapus, "Swarm metrics in peer-to-peer systems," in *Proceedings of the 9th RoEduNet IEEE International Conference*, 2010, pp. 276–281.

[6] M. Bardac, G. Milescu, and R. Deaconescu, "Monitoring a BitTorrent Tracker for Peer-to-Peer System Analysis," in *Intelligent Distributed Computing*, 2009, pp. 203–208.

[7] R. Deaconescu, M. Sandu-Popa, A. Draghici, and N. Tapus, "Using Enhanced Logging for BitTorrent Swarm Analysis," in *Proceedings of the 9th RoEduNet IEEE International Conference*, 2010, pp. 52–65.

[8] K. Harumoto, S. Fukumura, S. Shimojo, and S. Nishio, "A Location-Based Peer-to-Peer Network for Context-Aware Services in a Ubiquitous Environment," pp. 208–211, 2005.

[9] W. Thiengkunakrit, S. Kamolphiwong, T. Kamolphiwong, and S. Saewong, "Enhanced Context Searching Based on Structured P2P," pp. 309–313, April 2010.

[10] C. Doulkeridis, V. Zafeiris, and M. Vazirgiannis, "The role of caching and context-awareness in P2P service discovery," *International Conference On Mobile Data Management*, pp. 142–146, 2005.

[11] K. F. Yeung, Y. Yang, and D. Ndzi, "A Context-Aware System for Mobile Data Sharing in Hybrid P2P Environment," pp. 63–68, 2009.

[12] M. Stonebraker, "SQL databases v. NoSQL databases," *Communications of the ACM*, vol. 53, no. 4, pp. 10–11, Apr. 2010.

[13] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data Warehousing and Analytics Infrastructure at Facebook," pp. 1013–1020, 2010.

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

[15] Redis Community, *Redis Documentation*, accessed in March 2011, http://redis.io/documentation.

[16] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Computing Surveys*, vol. 31, no. 3, pp. 264–323, Sep. 1999.