A fault-tolerant approach to storing objects in distributed systems

Ciprian Dobre*, Florin Pop*, Valentin Cristea*
*Computer Science Department, Faculty of Automatic Control and Computes Science,
University POLITEHNICA of Bucharest
E-mails: {ciprian.dobre, florin.pop, valentin.cristea}@cs.pub.ro

Abstract

Over the Internet today, computing and communications environments are more complex and chaotic than classical distributed systems, lacking any centralized organization or hierarchical control. Peer-to-Peer network overlays provide a good substrate for creating large-scale data sharing, content distribution and application-level multicast applications. We present DistHash, a P2P overlay network designed to share large sets of replicated distributed objects in the context of large-scale highly dynamic infrastructures. The system uses original solutions to achieve optimal message routing in hop-count and throughput, provide an adequate consistency among replicas, as well as provide a fault-tolerant substrate. In this we present result proving that the system is able to scale to a large number of nodes, and it includes the fault tolerance and system orchestration mechanisms, added in order to assess the reliability and availability of the distributed system in an autonomic manner.

1. Introduction

Large scale distributed systems are hardly ever "perfect". Due to their complexity, it is extremely difficult to produce flawless designed distributed systems. While until recently the research in the distributed systems domain has mainly targeted the development of functional infrastructures, today new requirements have emerged for large scale distributed systems; among these requirements, fault tolerance is needed by more and more modern distributed applications, not only by the critical ones. The clients expect them to work despite possible faults occurring.

Peer-to-peer (P2P) systems have been widely adopted for communication technologies, distributed system models, applications, platforms, etc. Currently there has been much interest in P2P network overlays (distributed systems that are not under any hierarchical organization or centralized control) because they provide (to some extent) a long list of features: selection of nearby peers, redundant storage, efficient search/location of data items, data permanence or guarantees, hierarchical naming, security. DHT-based systems [1] are an important class of P2P routing infrastructures that support the rapid development of a wide variety of Internet-scale applications, ranging from distributed file and naming systems to application-layer multicast. They enable scalable, wide-area retrieval of shared information. DHT-based networks have been widely utilized for accomplishing efficient resource discovery [2] for grid computing systems, as it aids in resource management and scheduling of applications. Recent advances in the domain of decentralized resource discovery have been based on extending the existing DHTs with the capability of multi-dimensional data organization and query routing.

In this we present DistHash, a system that combines capabilities of DHT networks with modern solutions to ensure fault-tolerance. It uses original solutions to achieve optimal message routing in hop-count and throughput, provide an adequate consistency among replicas, as well as provide a fault-tolerant substrate. In this we present result proving that the system is able to scale to a large number of nodes. It also includes the fault tolerance and system orchestration mechanisms, added in order to assess the reliability and availability of distributed systems in an autonomic manner.

The system is based on the implementation of a DHT (distributed hash table) in which a) peers do not equally participate in hosting published data objects; b) peers can join or leave the network at any time, without prior knowledge; c) the underlying network infrastructure can be different than the adopted communication scheme.

The rest of this paper is organized as follows. Section 2 presents related work. The architecture of the DHT system is presented in section 3. In Section 4 and 5 we present implementation details, together with the

presentation of the proposed concepts and algorithms. In Section 6 we present results demonstrating the validity and performances of the proposed solution. Finally, in Section 7 we present conclusions and future work.

2. Related Work

Distributed Hash Table (DHTs) networks are decentralized distributed solutions that partition the keys inserted into the hash table among the participating nodes. They usually form a structured overlay network in which each communicating node is connected to a small number of other nodes. Any DHT could be easily turned into a system offering communication services.

The Content Addressable Network (CAN) is a distributed decentralized P2P infrastructure that provides hash-table functionality on Internet-like scale [3]. CAN is designed to be scalable, fault-tolerant, and self-organizing. Unlike other solutions, the routing table does not grow with the network size, but the number of routing hops increases faster than logN. Still, CAN requires an additional maintenance protocol to periodically remap the identifier space onto nodes. Recently several solutions were proposed [4] to address the problem of fault-tolerance in such a network.

Pastry [5] is a scalable, distributed object location and routing scheme based on a self-organizing overlay network of nodes connected to the Internet. Pastry performs application-level routing and object location in a potentially very large overlay network of nodes connected via the Internet. Pastry is completely decentralized, scalable, and self-organizing; it automatically adapts to the arrival, departure and failure of nodes. Fault tolerance among members of the distribution tree is accomplished through the use of timeouts and keepalives with actual data transmissions doubling as keepalives to minimize traffic. Sharing similar properties as Pastry, Tapestry [6] employs decentralized randomness to achieve both load distribution and routing locality. The difference between Pastry and Tapestry is the handling of network locality and data object replication. Tapestry's architecture uses variant of the distributed search technique, with additional mechanisms to provide availability, scalability, and adaptation in the presence of failures and attacks. For fault tolerance, nodes keep c secondary links such that the routing table has size $c * B * log_B N$.

The Chord protocol [7] uses consistent hashing to assign keys to its peers. Consistent hashing is designed to let peers enter and leave the network with minimal interruption. It is closely related to both Pastry and Tapestry, but instead of routing towards nodes that share successively longer address prefixes with the destination, Chord forwards messages based on numerical difference with the destination address. Although Chord adapts efficiently as nodes join or leave the system, unlike Pastry and Tapestry, it makes no explicit effort to achieve good network locality.

The Kademlia [8] network assigns each peer a NodeID in the 160-bit key space, and (key,value) pairs are stored on peers with IDs close to the key. A NodeID-based routing algorithm is used to locate peers near a destination key. One of the key architecture of Kademlia is the use of a novel XOR metric for distance between points in the key space.

The Viceroy [9] decentralized overlay network is designed to handle the discovery and location of data and resources in a dynamic butterfly fashion. Even if its diameter of the overlay is better than CAN and its degree is better than Chord, Tapestry or Pastry, Viceroy has no built in support for fault tolerance.

Currently there are limited attempts to approach the problem of fault-tolerance in previously proposed DHT networks. Unlike our solution, these attempts to introduce fault-tolerance in these systems are based on best-effort approaches, where an arbitrarily large fraction of the peers can reach an arbitrarily large fraction of the data items.

3. DistHash's Architecture

The architecture of the proposed system is presented in Figure 1. The main components are the peers (called Agents) and the super-peers (called RAgents). The Agents are responsible for storing local replicas of the currently registered distributed objects. RAgents are also Agents (they have local replicas), but also contain the meta-data catalogues. In this architecture a subset of Agents is under the control of a RAgent. This leads to forming a cluster of peers. The RAgents are interconnected, forming a complete graph of connections. All control messages are routed between RAgents. Thus, the RAgents connect together several connected clusters of Agents.

Such a hierarchical interconnection topology ensures scalability and, as we will describe later on, an optimal fault-tolerance necessary for modern critical data-intensive applications. It also reflects the real-world phenomenon of large Internet-scale systems. As demonstrated in [10], networks as diverse as the Internet tend to organize themselves so that most peers

have few links while a small number of peers have a large number of links.

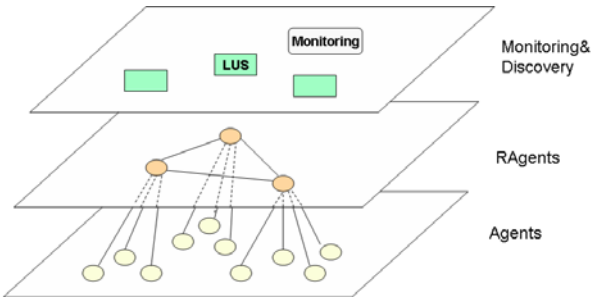

Fig. 1. DistHash's architecture.

In addition, DistHash also includes several lookup&discovery services, distributed in different locations. These services store information regarding the currently existing RAgents. Therefore, when a new cluster is formed, its RAgent dynamically registers to the closest lookup service. A client then uses one lookup service to obtain the of-interest information.

The grouping of Agents into clusters is based on their geographic positions, as well as a set of condition metrics. The idea was previously been validated in [11], where we proposed an algorithm to assist the EVO P2P videoconferencing system. In this approach peers dynamically detect the best reflectors to which to connect to. In the algorithm the best reflectors are chosen based on their network and geographic location (Network domain, AS domain, Country, Continent), and also based on their current load values, number of currently connected clients, current network traffic.

When an Agent joins the network, it researches the list of RAgents from one of the lookup services. It then estimates the best RAgent to which to connect. The RAgent is chosen based on the network loads, geographic position and the number of other Agents connected to the RAgents. In the end the Agent computes a metric such that it connects to one of the closest RAgent having the smaller number of Agents connected.

The clusters are formed based on the way modern pending event structures (PES) are built. As indicated in [12], a Calendar Queue is one of the most advanced PES structure, built by dividing the queue into buckets. Similarly, we consider that we have a number of RAgents (the equivalent of a bucket) connecting several Agents (having a length of the bucket or bucket-width). In this approach it is critical to optimally chose the number of Agents connected to a RAgent and the number of RAgents existing in the system. If the number of Agents is higher compared to the number of RAgents, the meta-data catalogue grows too big and the operations on it require a longer time to complete. On the other hand, if the number of Agents is low compared to the number of RAgents, the number of control messages required to access information from the system is higher. These situations are illustrated in Figure 2.

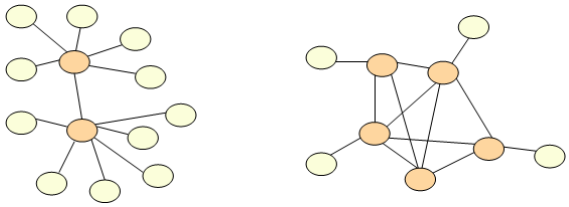

Fig. 2. The clustering when there are few RAgents (left) or too many RAgents (right).

To mediate these problems, the number of RAgents increases and decreases as the number of Agents grows (new Agents join the system) and shrink (Agents leave the system). Whenever the number of Agents connected to one RAgent becomes too high, a new RAgent is promoted from Agents (based on a voting algorithm), and the cluster is divided in two, by splitting the remaining Agents between the two RAgents. The meta-data catalogue is also divided between the two RAgents, and the lookup information is updated accordingly. This is illustrated in Figure 3.

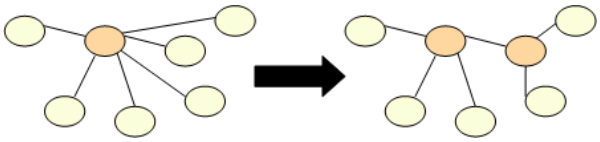

Fig. 3. The creation of a new cluster.

Whenever the number of Agents connected to a RAgent is too low, the cluster is destroyed by joining the Agents with the ones from an adjacent cluster. The meta-data catalogue is also merged within the catalogue of the new cluster's RAgent, and the lookup information is again updated accordingly.

The two operations involve exchange of several control messages to update the current status of clusters, as well as the possible reconnection of several Agents to new RAgents and RAgents with other RAgents. In order to minimize the number of costly operations, we started from the threshold values for the splitting and joining operations observed in [12]. But these values are also dynamically adjusted accordingly at runtime.

Inside a cluster, peers connect as represented in Figure 4. Because the RAgent stores the meta-data catalogue, it must not represent a central-point-of-

failure. For this, one specially designated Agent from the cluster also acts as secondary backup replica for the RAgent. In case of failure the secondary Agent takes over the role of the RAgent. All updates of the meta-data catalogue will immediately also propagate to the secondary RAgent replica.

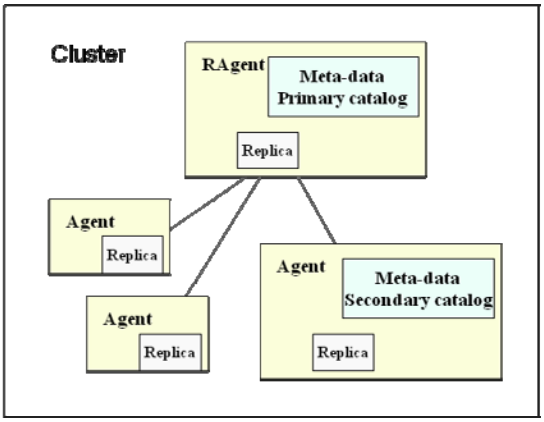

Fig. 4. Roles of peers inside a cluster.

4. Data Management

The main role of the presented system is to ensure a fault-tolerant distributed storage space for objects. For this, each Agent maintains a copy of several replicated objects. These are Java serializable objects that can represent anything, from basic data types to blobs or data files.

The data consistency among replicas is ensured by a simple strategy that requires that the lock associated with each object is maintained by one unique owner an Agent. The ownership, as well as information regarding current objects existing inside a cluster, are maintained within the meta-data catalogue (see Figure 5).

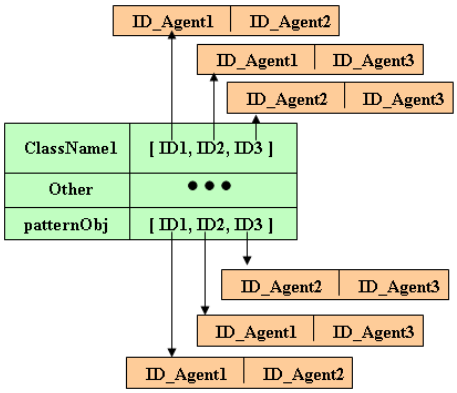

Fig. 5. Information stored in the meta-data catalogue.

The meta-data catalogue is organized as a hash table, where the keys represent specific information such as the class name, an access pattern or information declared by the user creating a new object as being of interest. The values represent linked-lists of (key, value) mappings. The key in the linked list represents an object identifier matching a particular pattern key (ID1, ID2, etc.). An object identifier is a hash uniquely identifying a particular object. Each value also points to a linked list of Agent identifiers (ID_Agent1, ID_Agent2, etc). The first identifier always points to the Agent currently having the ownership over a specific object.

Within a replica the objects are identified by a key representing the identifier of the object. In order to access a particular object a request travels to the RAgent, and from there to the Agent. The search operation on such a structure involves several stages (Figure 6).

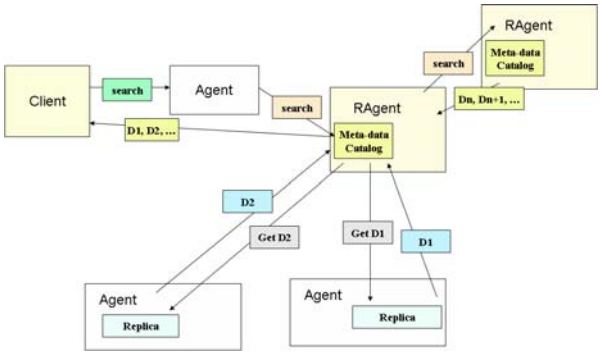

Fig. 6. The search operation.

First a client initiating a search operation contacts an Agent (the lookup services are secured and the clients do not have direct access to the internal information regarding the list of RAgents). The Agent then initiates the search process in the name of that client. It contacts the RAgent and it sends the current search criterion. The RAgent next finds the key within the meta-data catalogue corresponding to the search criterion. This operation requires in the worst case *M* steps, where *M* represents the number of keys in the meta-data catalogue. This is improved by considering separate catalogue for exact information (class names) and for search patterns. Searching for the objects corresponding to a particular class name requires only *log M* operations (using the Java mapping of hash keys for strings corresponding to the string values).

After finding particular entries corresponding to the search criteria, the search operation next involves

obtaining the unique IDs of corresponding objects, as well as an Agent ID owning a particular object. This requires *2P* steps, where *P* is the number of objects matching the search criteria. The RAgent next contacts the Agent and obtains copies of each object. This involves again *2P* messages being sent and received (in fact, the RAgent will not send one request for each object detain by a particular Agent, but groups the requests into one and send it once – the actual number of exchanged messages is smaller than *2P*). The agent gets a particular object corresponding to a particular id in *log L* steps (a hash search), where *L* represents the number of objects currently possessed by an Agent. In the end, the data is collected and sent back to the Agent that initiated the search operation, and sent back to the Client.

Among RAgent the search operation involves several steps. The RAgent sends a request with a particular search criterion. Each RAgent respond back with the list of objects matching the criteria, together with their corresponding object identifiers. The first RAgent merges the received lists of objects, together with its own list. Considering *R* RAgents, the total number of steps involved by the search is $S = R * M * (4P+\log L)$.

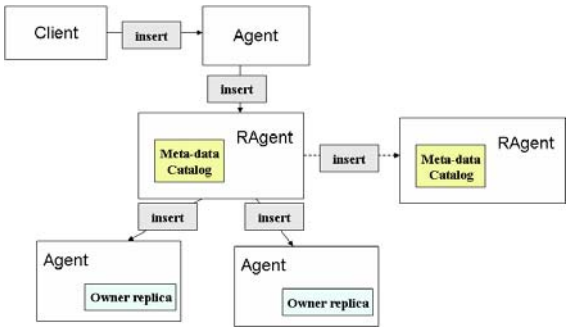

Fig. 7. The insert operation.

The insert operation involves several steps. First, a client willing to add a new object in the system contacts an Agent. The Agent then sends the insert request to the RAgent to which it is connected. The RAgent selects two Agents that will further keep two replicas of the object stored locally and it updates the meta-data catalogue with the information regarding the new object. It selects one Agent as being the owner of the object and sends the replicated objects to the two selected Agents. The selection is done such that to permanently balance the number of objects kept by each Agent. For this the RAgent maintains the current number of objects stored by each local Agent. When adding a new object, the RAgent selects the Agents with the fewest objects, but also being close to the client introducing them in the system. This approach guarantees a load balance of the Agents and it ensures a fault tolerant environment.

Assuming an ideal equilibrated split of the number of objects detained by all Agents in the system and considering that the search criteria is the class names of the objects, $M = B / R$, where *B* represents the number of objects at a particular moment kept in the entire system (the objects would be equally divided among clusters). The number of objects matching a particular class name would then be $P = 1$. Also, the number of objects kept by an Agent is given by $L = (2B / R) / (N / R)$, meaning the total number of objects kept inside a cluster $(2B / R)$ divided by the number of Agents in a cluster (N / R), where N represents the total number of Agents. In the end, $S=B*(4+\log 2B/N)$. This means that, as the number of objects increases, the complexity of the search operation increases, but in the same time when the number of Agents increase the complexity of the search operation decreases.

In reality, using this approach the objects could in fact accumulate more around a single cluster. To cope with situation the RAgent can also delegate the insert operation, if the size of the local catalogue grows too big, to other RAgents. This is illustrated in Figure 7.

To further optimize the search operation, a special query search message is implemented with the role of returning the first object matching a particular criterion. When the RAgent receives too many searches for a particular object, located in another cluster, it can decide to also move the object inside the local cluster. The decision is based on successive searches returning same objects. In this way duplicate searches does not travel anymore to all RAgents.

The read operation can also be performed on any Agent having a replica of a particular object. A special request search syntax returns back the address of an Agent having a particular replica object. The client can then initiate successive read operations on that particular replica, directly accessing one particular Agent. This greatly reduces the number of steps involved and the client has a greater flexibility in using the system.

The update operation is an asynchronous operation, where an Agent initiates an update operation on a particular object. The RAgent keeps a lock on the owner of the object and then, asynchronously, updates all active replicas of that objects. As soon as the RAgent receives the request and performs the lock it sends back a notification message that the update operation is in progress. This approach ensures consistency among the replicas, as no two updates would happen simultaneous on different replicas

because only one owner of a particular object exists in the system.

5. Fault Tolerance Management

In the described architecture, faults are always detected. For example, when a client initiates an update operation the RAgent might discover that a particular Agent is no longer active in the system. In addition, the Agents and RAgents periodically send heartbeat messages among them. The period is chosen such as to be high enough not to introduce much communication overhead in the system.

Whenever an Agent fails, the RAgent in charge of its local cluster initiates a process of object reallocation (illustrated in Figure 8). First, the ownerships of objects previously owned by the Agent who failed are inherited by the surviving Agents having replicas of that objects. This approach gives penalties to failing Agents and rewards the robust Agents. The Agents running for a longer time eventually become owners of more and more objects.

Next the RAgent selects one Agent to be the new keeper of the secondary replica of an object previously detain by the crashed Agent. The algorithm used to select the Agent considers a balancing of the number of objects hold between Agents at all times.

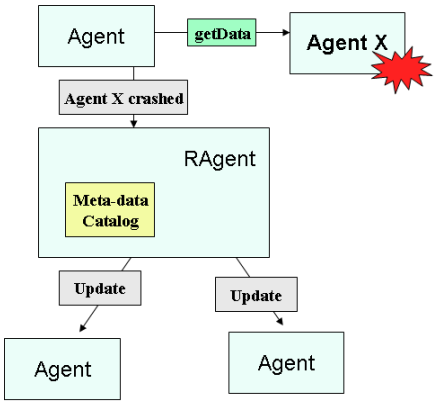

Fig. 8. A crash scenario of an Agent.

When the RAgent fails, the secondary RAgent replica takes its place. When this happens, the replica updates the corresponding entry in the lookup service, and it connects to all other RAgents. Among the remaining Agents one is selected (based on a voting procedure) to become the new RAgent replica (see Figure 9).

The voting procedure is consistent, such that if two agents observe the disappearing of the same RAgent they both initiate the voting procedure, but in the end both voting processes lead to the same Agent being selected to be the new RAgent replica.

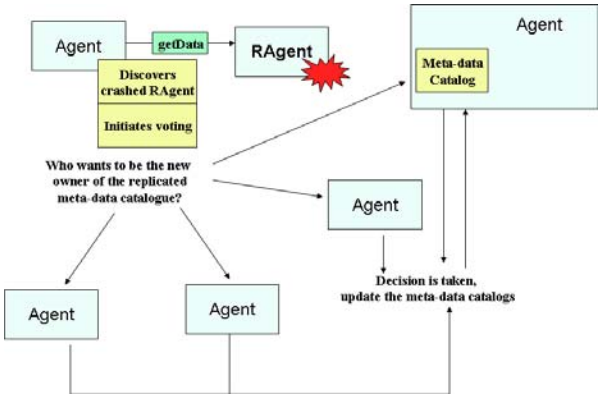

Fig. 9. A crash scenario of an RAgent.

In the event of a transient failure (an Agent or RAgent temporarily leaving the system and then entering back at a later time) the RAgent (newly or one from another cluster) detects the situation and informs the newly entered peer of the changed. In this way, we can cope with both transient and permanent failures of nodes in the system. Also, the system is highly resilient to faults as allows a very large number of nodes to fail without losing information. In addition, the information is rapidly replicated so that to offer a maximum degree of fault tolerance level to clients.

6. Results

The system was evaluated using a cluster of Intel Xeon E5405 quad-core stations, connected through Gigabit Ethernet. For the first experiments we used 9 Agents and inserted objects in various points in the systems. At some point we simulated a crash of different Agents in the system.

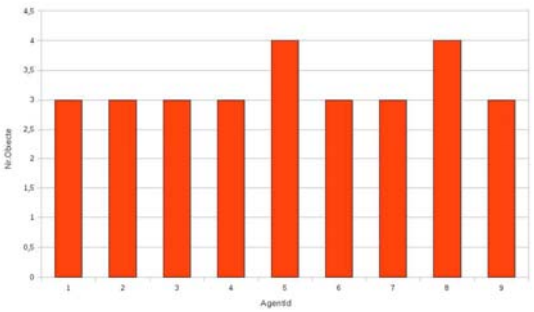

Fig. 10. The distribution of replicated objects after the crash.

Figure 10 presents the evolution of the objects as a result of such a crash. The system automatically re-execute replication such that always to preserve at least two replicas of the same object.

After evaluating the correctness of the proposed solution, we executed further tests to demonstrate also its performances. For this we executed a series of experiments that evaluated the behavior of the system under various changes (such as changes in the topology or faults in various links). For monitoring the performances of the system we used a dedicated monitoring system named LISA [11]. We were particularly interested in the load of the workstations on which the Agents were executed (as a measure of the balancing of activities among Agents), the processor and memory usages and the load of the networks connecting the Agents. We executed different experiments by varying the number of Agents (up to 20) and the size of the clusters being formed was established at 10.

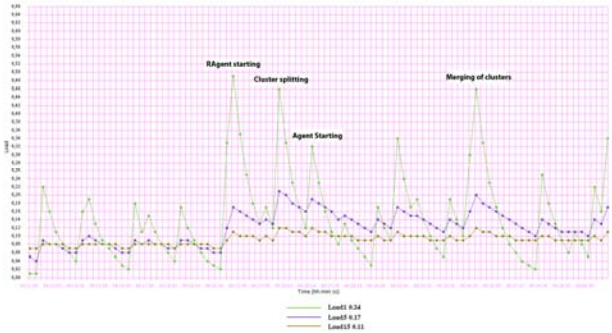

Fig. 11. The load on a test machine.

Figure 11 demonstrates the load of one of the testing machine. The topology changes (joining and dividing clusters) affect the system mostly. However, these changes last for little time and the system stabilizes quickly.

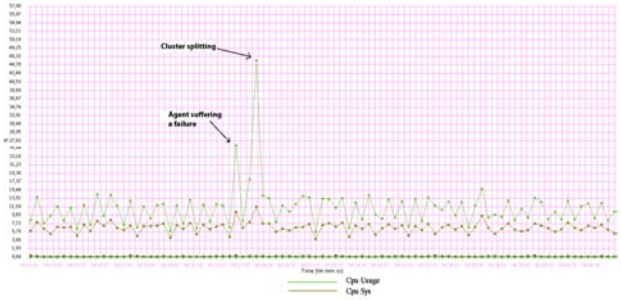

Fig. 12. The CPU usage on a test machine.

The CPU usage (see Figure 12) was around 8%. The spikes appeared because of the event of the exit of an Agent, followed by immediate cluster division. As seen, the joining operation does not affect to a vast extend the CPU usage.

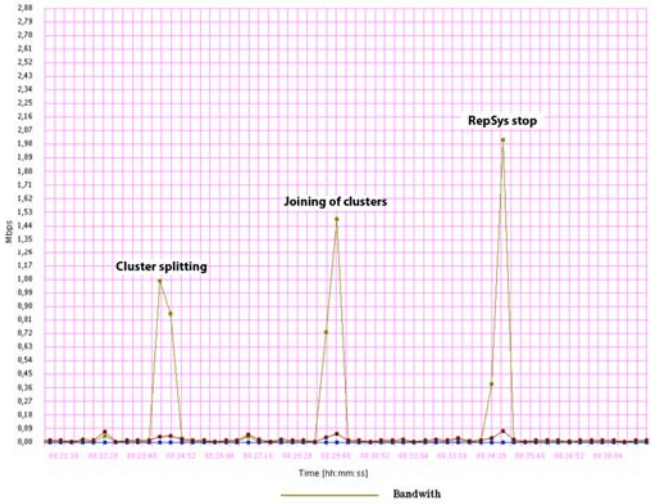

Fig. 13. The generated network traffic.

Finally, the network traffic being generated (see Figure 13) is low. The topology changes are, again in this case, the most costly operations. However, as noticed, the system quickly recovers.

These experiments demonstrate the performance of the presented system. As presented, the system is capable of quickly adapting to various crashes or Agents dynamically entering or exiting.

7. Conclusions

In this paper we presented DistHash, a P2P overlay network designed to share large sets of replicated distributed objects in the context of large-scale highly dynamic infrastructures. We presented the original adopted solutions to achieve optimal message routing in hop-count and throughput, provide an adequate consistency approach among replicas, as well as provide a fault-tolerant substrate.

The system provides a middleware level functionality to users needing to use a shared memory concept system to be accessed by distributed application running on an Internet-like large-scale infrastructure. DishHash is highly robust in the face of failures happening in various points of the system, as well as peers highly dynamically entering or leaving the system. It also offers a balanced approach to the way replicated objects are kept inside the system. The operations are optimized in terms of steps required, as well as number of messages exchanged between peers. For that we consider a hierarchical approach in which

we organize peers in clusters. Inside each cluster the functionality is almost autonomous so that the total number of peers would not influence the performance of the system. This means that DistHash also offers scalability to end-users.

In this we presented results proving that the system is able to scale to a large number of nodes, and it includes the fault tolerance and system orchestration mechanisms necessary to assess the reliability and availability of the distributed system in an autonomic manner. In the future we plan to further extend the system with more advanced routing capabilities, to make better use of the meta-data catalogue in order to generate search queries more complex and we plan to optimize the overall performances of the system.

Acknowledgments

The research presented in this paper is supported by national project "DEPSYS – Models and Techniques for ensuring reliability, safety, availability and security of Large Scale Distributes Systems", Project "CNCSIS-IDEI" ID: 1710.

8. References

[1] A.B. Smith, C.D. Jones, and E.F. Roberts, "Article Title", *Journal*, Publisher, Location, Date, pp. 1-10.

[2] Jones, C.D., A.B. Smith, and E.F. Roberts, *Book Title*, Publisher, Location, Date.

[1] K. Lua, J. Crowcroft, M. Pias, J. Sharma, S. Lim, "A survey and comparison of peer-to-peer overlay network schemes". *IEEE Communications Surveys & Tutorials,* 7(2), pp. 72-93, 2005.

[2] R. Ranjan, L. Chan, A. Harwood, S. Karunasekera, R. Buyya, "Decentralised Resource Discovery Service for Large Scale Federated Grids". In *Proceedings of the Third IEEE international Conference on E-Science and Grid Computing*, pp. 379-387, December 10 - 13, 2007.

[3] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, "A scalable content-addressable network", in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications (SIGCOMM '01*), San Diego, California, United States, pp. 161-172, 2001.

[4] J. Saia, A. Fiat, S.D. Gribble, A.R., Karlin, S. Saroiu, "Dynamically Fault-Tolerant Content Addressable Networks", In *Revised Papers From the First international Workshop on Peer-To-Peer Systems*, (P. Druschel, M. F. Kaashoek, A. I. Rowstron, Eds.), Lecture Notes In Computer Science, vol. 2429. Springer-Verlag, London, pp. 270-279, March 07 - 08, 2002.

[5] A. Rowstron, P. Druschel, „Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems", in *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms* (Middleware'01), 2001.

[6] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, J.D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment", *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, 2004.

[7] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications", *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.

[8] P. Maymounkov, D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric", in *Revised Papers From the First international Workshop on Peer-To-Peer Systems,* (P. Druschel, M. F. Kaashoek, A. I. Rowstron, Eds.) Lecture Notes In Computer Science, vol. 2429. Springer-Verlag, London, pp. 53-65, March, 2002.

[9] D. Malkhi, M. Naor, D. Ratajczak, "Viceroy: a scalable and dynamic emulation of the butterfly", in *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing* (PODC '02)*,* Monterey, California, pp. 183–192, July 2002.

[10] A. Barabasi, R. Albert, H. Jeong, G. Bianconi, "Power-law distribution of the world wide web", *Science*, vol. 287, 2115a, 2000.

[11] C. Dobre, R. Voicu, A. Muraru, I.C. Legrand, "A Distributed Agent Based System to Control and Coordinate Large Scale Data Transfers", in *Proc. of the 16th International Conference on Control Systems and Computer Science* (CSCS-17), Bucharest, Romania, 2007.

[12] R. Brown, "Calendar Queues: A fast O(1) Priority Queue implementation for the simulation event set problem". *Comm. ACM*, 31 (10), pp. 1220-1227, 1988.