# BackupIT: An Intrusion-Tolerant Cooperative Backup System

Sérgio Loest, Marcelo Madruga, Carlos Maziero
Graduate Program in Computer Science
Pontifical Catholic University of Paraná, Brazil
{sloest,mcmadruga,maziero}@ppgia.pucpr.br

Lau Lung
Graduate Program in Computer Science
Federal University of Santa Catarina, Brazil
lau.lung@inf.ufsc.br

## Abstract

*Reliable storage of large amounts of data is always a delicate issue. Availability, efficiency, data integrity, and confidentiality are some features a data backup system should provide. At the same time, corporate computers offer spare disk space and unused networking resources. In this paper, we propose an intrusion-tolerant cooperative backup system that provides a reliable collaborative backup resource by leveraging these independent, distributed resources. This system makes efficient use of network and storage resources through compression, encryption, and efficient verification processes. It also implements a protocol to tolerate Byzantine behaviors, when nodes arbitrarily deviate from their specifications. Experiments performed to evaluate the proposal showed its viability.*

## 1 Introduction

Data storage capacity grows at high rates, but the amount of data to be stored grows in the same proportion [12]. Data storage systems should ensure data availability and integrity for large amounts of data. A key component to ensure such requirements is the data backup system. The storage size needed for backup copies in a corporation may easily reach Terabytes. At the same time, there is a large amount of free storage space in individual computers: in average, 50% of a corporate computer disk is unused [3]. There is also a fair amount of idle processing power, as the most frequent activity of a corporate computer is to wait for keyboard input. As most computers in a corporation are connected to a local network, this environment is favorable for the use of a cooperative backup system.

Cooperative backup systems [15] use peer-to-peer technology to deploy a non-hierarchical distributed backup environment, in which each node uses its free disk space to backup data from others nodes, without a central backup resource. Such distributed backup environments should ensure the same availability, integrity, and confidentiality requirements as conventional systems do, while incurring in lower deployment and operation costs. In this paper, we propose an Intrusion-Tolerant Cooperative Backup System that uses a Byzantine Quorums System (*BQS*) [17]. Byzantine quorums systems are known by their availability and efficient use of replicated data, even in the presence of malicious nodes. Thus, this work aggregates the advantages of quorum systems to a cooperative backup system, to provide a correct operation even in the presence of malicious nodes.

This paper is organized as follows: Section 2 provides an overview of cooperative backup systems; Section 3 reviews the main Byzantine Quorums concepts; in section 4, the overlay peer-to-peer substrate Pastry is briefly presented; Section 5 presents our proposal and details its main aspects; some experimental results are discussed in section 6; Section 7 discusses some related work, and finally Section 8 draws some conclusions and future work.

## 2 Cooperative Backup Systems

Cooperative backup systems (*CBS*) [15] use peer-to-peer (*p2p*) technology to deploy a non-hierarchic distributed backup environment. In such systems, each node participates by giving part of its local hard disk to backup data from other peers, and vice versa [1]. This way, as new nodes join the backup system, the demand for backup increases, but the total storage capacity of the system also increases accordingly. The distributed nature of *p2p* networks also increases robustness in case of failures, by allowing the replication of data over multiple peers, and by enabling peers to find the data without relying on a central index server.

The implementation of a non-hierarchical distributed cooperative backup service without prior trust relationship among nodes is not a trivial task. Some threats must be taken in account, like nodes with selfish behavior (refusing to cooperate), peers that may fail, and malicious peers that may attack the data reliability or the service availability. Thus, cooperative backup services must implement mechanisms to provide data integrity and consistency, service availability, data privacy, and trust management [12].

In friend-to-friend (*f2f*) backup systems [14], the *p2p* system is formed by a social network. The presence of malicious peers is minimized, due to trust relationships set through the social network. This allows the use of fewer replicas, reducing demands in network bandwidth and storage area. Other techniques can be used to detect/tolerate malicious nodes, like periodic random-block challenges, to certify that other peers still hold the backup files [15].

## 3 Byzantine Quorum Systems

A Byzantine Quorums System (BQS) [17] is a replicated data storage system that ensures data integrity and availability even in the presence of arbitrary faults in some of its replicas [7]. The protocols used in a BQS provide termination guarantee and thus do not require agreement protocols to ensure consistency among replicas. This means that they are not susceptible to the FLP impossibility [8] and can be implemented in asynchronous systems.

Conceptually, a BQS can be defined as a set of subsets of nodes in a distributed system. Each node subset is called a *quorum* and intersects with all other quorums in the system. According to the availability property, there is at least one quorum in the system that is formed by correct nodes only. The intersection properties guarantee that the transactions performed in different quorums maintain data consistency. The use of quorums is a method to increase availability and integrity of replicated data, because each quorum can act on behalf of the entire system, thereby increasing its availability and performance.

In a BQS, clients perform read and write operations on registers replicated over a quorum of nodes. Registers can be signed by the client, being then called *auto-verifiable*; the key owner can detect non-authorized content changes by a malicious server. A typical BQS using auto-verifiable registers to survive up to $f$ failures uses groups of $3f + 1$ replicas with quorums of size $2f + 1$. The quorum intersections will have size $f + 1$, ensuring that the intersection of any two quorums has at least one correct replica [16].

## 4 The Pastry Environment

Pastry [18] is a substrate for peer-to-peer systems providing location and routing services. It builds an auto-organized overlay network for distributed applications. It is decentralized, fault resistant, scalable, reliable, and has good message routing properties. In Pastry, nodes and objects receive 128-bit unique identifiers, respectively called *nodeId* and *key*. Messages are routed to a node which *nodeId* is closest to a given key using $O(\log N)$ steps, where $N$ is the number of nodes in the overlay network [9].

Each Pastry node keeps track of its immediate neighbors in the identifier space and notifies the local application about new nodes, failures and recoveries. As the *nodeIds* are randomly assigned, the set of nodes with adjacent *nodeIds* is potentially diverse in geography, ownership, and jurisdiction. An heuristic ensures that, among a set of nodes with the closest *nodeIds* to a given key, the message is likely to first reach the node nearest to the sender node. The routing data maintained by each node consists in a leafset ($\mathbb{L}$) and a routing table. The leafset contains $|\mathbb{L}|/2$ neighbor *nodeIds* of the local node (usually 32 or 64 inputs), sorted by identifier proximity. Pastry updates the routing information whenever a node joins or leaves the overlay network, keeping the routing costs logarithmic [18].

The Pastry API functions relevant for this work are `pastryInit()`, allowing a node to join a Pastry network, and `route(msg,key)`, to route a message to the node whose *nodeId* is numerically closest to *key*. The applications built on Pastry must export the following callbacks:

- `deliver(msg,key)`: called by Pastry when a message is received and the local node's *nodeId* is numerically closest to *key*;
- `forward(msg,key,nextId)`: called by Pastry just before a message is forwarded to the node with *nodeId = nextId*.
- `newLeafs(leafSet)`: called by Pastry whenever the node's *leafset* changes.

## 5 BackupIT - An Intrusion-Tolerant Cooperative Backup System

The BackupIT system uses the resources available in an *Intranet* (a set of computers under a single administrative entity) to store data backups collaboratively. Each participant computer stores data backups for other local computers. This way, a better use of the available disk space, processing power, and network bandwidth is obtained. Availability, intrusion tolerance, and data integrity are achieved through data replication among the participating nodes. Replication uses Byzantine Quorum Systems, avoiding single failure points. The next sections describe the system architecture, its main components, and the algorithms used in its operation.

### 5.1 System architecture

The system consists of a set of nodes connected to a *p2p* overlay network. All nodes have the same functions, forming a pure (non-hierarchical) *p2p* network. The nodes use *BQS* protocols to provide backup operations. Each node is responsible to reply requests, to store/retrieve data for/from

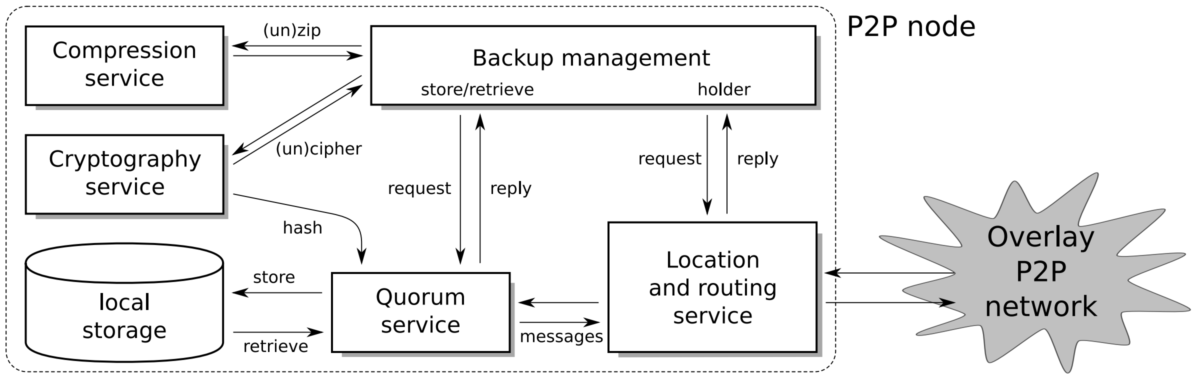

**Figure 1. System Architecture**

other nodes, and to provide an interface to the local applications. Each node has five local components: *Backup Management*, *Quorum Service*, *Location and Routing Service*, *Compression Service*, and *Cryptographic Service* (figure 1).

The *Backup Management* handles client requests to backup/retrieve files. It is responsible for the generation of cryptographic signatures (*hashes*), file storing, and integrity verification during file retrieval. The *Quorum Service* provides intrusion tolerance, availability, and data integrity. It is presented with more detail in section 5.2. The *Location and Routing Service (LRS)* is responsible for the nodes/files unique identifiers generation and location, message routing, and for the definition of the nodes in each node's quorum system. This service is provided by the Pastry substrate (section 4). Files to backup are compressed using the *Compression Service*, to reduce the storage space and network bandwidth needs. The *Cryptographic Service* uses symmetric cryptography to ensure data confidentiality and integrity.

A data expiration policy was adopted to discard obsolete data: each backup stored in the system has a validity period associated to it, defined by its owner. The backup is then kept during that period, and can be discarded after it.

## 5.2 Quorum service

This project uses the *f-dissemination* Byzantine Quorum System described in [17]. This quorum system stores self-verifying data and uses symmetrical quorums (read and write quorums have the same size). It can be built with a smaller number of nodes and its protocols demand a reduced number of messages, fitting well to Intranets. The consistency semantics of this quorum system is regular: if there is no concurrent writes, a read operation always returns the last written value; otherwise, it returns one of the written values. As backup systems usually have a single writer semantics, there are no concurrent writes.

Each system node $p_i$ builds a quorum system $\mathbb{S}_i$ with $|\mathbb{S}_i| \geq 3f+1$ nodes, where $f$ is the maximum number of faulty nodes, defined during the quorum system formation. The nodes that form the quorum system of a given node are defined from its *leafset* $\mathbb{L}_i$, given by the location and routing service. Quorum systems of distinct nodes overlap; therefore, each node will belong to several quorum systems simultaneously. If a quorum system has $3f+1$ members, each node in it will have other $3f$ nodes in its quorum system; if all quorum systems are distinct, then each node will be part of $3f$ distinct quorum systems.

## 5.3 System model

An asynchronous system is assumed. The cooperative backup system is composed by a set $\mathbb{P}$ with $N$ nodes or peers $p_1 \ldots p_N$. The number of faulty nodes tolerated by the system is $f$, with $3f+1 \leq N$. Each node $p_i$ is uniquely identified and located in system by a node identifier $nodeId_i$. The *leafset* of node $p_i$, defined by the *Pastry* substrate, is $\mathbb{L}_i$. Each node $p_i$ builds a byzantine quorum system $\mathbb{S}_i$ using the first $3f$ nodes of its leafset $\mathbb{L}_i$ and itself. A quorum $\mathbb{Q}_i^*$ is any subset of $\mathbb{S}_i$ with $2f+1$ nodes, that is, $\mathbb{Q}_i^* \subset \mathbb{S}_i$ and $|\mathbb{Q}_i^*| = 2f+1$.

A file to store in the backup system is indicated as $x$, and $name(x)$ is its name. Finally, it is considered that $hash(x)$ is a cryptographic signature function, $zip(x)$ is a data compression function and $\{x\}_k$ represents the ciphering of $x$ using a symmetric cryptographic key $k$ (each node $p_i$ has its own secret cryptographic key $k_i$).

The failure model assumes that up to $f$ nodes can deviate from their specifications. Faulty nodes can stop, omit messages, send false messages and incorrect replies. However, the cryptographic signatures allows to detect incorrect messages. Flooding attacks are out of the scope of this work. Stop failures and omissions are handled using *time-outs*. Faults are considered independent: the probability of a fault in one node is independent of the occurrence of a fault in other node. Finally, any node abnormal behavior detected generates an error notification for external intervention.

## 5.4 File storage and retrieval

Algorithm 1 indicates the actions of a node $p_i$ to store a file $x$. First, $p_i$ generates an unique key for the file, using the *hash* of $nodeId_i$ concatenated with the file name (line 1). Next, the location and routing service is queried to identify the node $p_j$ responsible for that key (line 2). Node $p_j$ will reply to $p_i$ informing its quorum system $\mathbb{S}_j$ (line 3). In the sequence, $p_i$ compresses and ciphers the file $x$ using its secret key $k_i$ (line 4). Node $p_i$ should then send the file to a set of nodes $p_k$ in $\mathbb{S}_j$ (line 8). Each reply informs the *hash* $h_k$ of the file received by $p_k$ (line 9). If this *hash* is equal to the *hash* of the file sent by $p_i$, the reply is correct (lines 10 and 16). If not, or if $p_k$ did not reply, an error is registered (line 11). Node $p_i$ waits for at least $2f + 1$ correct replies (line 7), or more than $f$ errors (line 12).

**Algorithm 1** Node $p_i$ stores file $x$

```
1:  $key \leftarrow hash(nodeId_i : name(x))$
2:  send $holderReq(key)$ to LRS
3:  receive $holderReply(\mathbb{S}_j)$ from $p_j$
4:  $x' \leftarrow \{zip(x)\}_{k_i}$
5:  $\mathbb{C} = \phi$ // nodes with correct reply
6:  $\mathbb{E} = \phi$ // nodes with error
7:  **while** $|\mathbb{C}| < 2f + 1$ **do**
8:      send $storeReq(x')$ to $p_k \in \mathbb{S}_j$
9:      receive $storeReply(h_k)$ from $p_k$ or *time-out*
10:     **if** *time-out* $\vee (h_k \neq hash(x'))$ **then**
11:         $\mathbb{E} \leftarrow \mathbb{E} \cup \{p_k\}$
12:         **if** $|\mathbb{E}| > f$ **then**
13:             **Error:** more than $f$ faulty nodes
14:         **end if**
15:     **else**
16:         $\mathbb{C} \leftarrow \mathbb{C} \cup \{p_k\}$
17:     **end if**
18: **end while**
```

It should be observed that algorithm 1 is not the same proposed in [17] for $f$-dissemination quorums with regular *multi-writer/multi-reader* semantics. This algorithm has an additional step, where client compares the *hashes* sent by the servers to the local *hash* (line 10). This step helps to detect file transmission problems.

Algorithm 2 shows the actions performed by a node $p_i$ to retrieve a previously stored file $x$. First, $p_i$ calculates the file key and locates the node $p_j$ responsible for it (lines 1 to 3). Next, $p_i$ requests the hash of the file identified by *key* to a quorum of nodes in $\mathbb{S}_j$ (lines 4 to 8). For each reply received, its hash $h_k$ is checked(lines 9 to 12). Node $p_i$ continues to pick nodes from $\mathbb{S}_j$ (lines 13 to 16) until it receives $2f + 1$ correct replies. Node $p_i$ then requests the file to one of the nodes that gave a correct reply, verifies if its *hash* is correct and return it to the user (lines 23 to 29). Otherwise, another node should be queried. According to the quorum theory [17], if there are up to $f$ faults, the algorithm will correctly retrieve the file.

**Algorithm 2** Node $p_i$ wants to retrieve file $x$

```
1:  $key \leftarrow hash(nodeId_i : name(x))$
2:  send $holderReq(key)$ to LRS
3:  receive $holderReply(\mathbb{S}_j)$ from $p_j$
4:  $\mathbb{P} \leftarrow \phi$ // nodes queried
5:  **for all** $p_k \in \mathbb{Q}_j^* \subset \mathbb{S}_j$ **do**
6:      send $hashReq(key)$ a $p_k$
7:      $\mathbb{P} \leftarrow \mathbb{P} \cup \{p_k\}$
8:  **end for**
9:  $\mathbb{R} \leftarrow \phi$ // nodes that answered
10: $\mathbb{T} \leftarrow \phi$ // nodes with *time-out*
11: **while** $(|\mathbb{R}| < 2f + 1) \wedge (|\mathbb{T}| \leq f)$ **do**
12:     receive $hashReply(h_k)$ from $p_k \in \mathbb{P}$ or *time-out*
13:     **if** *time-out* **then**
14:         $\mathbb{T} \leftarrow \mathbb{T} \cup \{p_k\}$
15:         send $hashReq(key)$ to $p_m \in (\mathbb{S}_j - \mathbb{P})$
16:         $\mathbb{P} \leftarrow \mathbb{P} \cup \{p_m\}$
17:     **else**
18:         $\mathbb{R} \leftarrow \mathbb{R} \cup \{p_k\}$
19:     **end if**
20: **end while**
21: $\mathbb{C} \leftarrow \{p_k \in \mathbb{R} \mid h_k = hash(x)\}$ // correct replies
22: **while** $\mathbb{C} \neq \phi$ **do**
23:     send $retrieveReq(key)$ to $p_k \in \mathbb{C}$
24:     receive $retrieveReply(x_k)$ from $p_k$ or *time-out*
25:     **if** *time-out* $\vee (hash(x_k) \neq hash(x))$ **then**
26:         $\mathbb{C} \leftarrow \mathbb{C} - \{p_k\}$
27:     **else**
28:         Decipher, uncompress and return $x_k$ to user
29:         Algorithm ends
30:     **end if**
31: **end while**
32: **Error:** no correct reply
```

From the algorithms, it can be inferred that the number of messages and the amount of data transferred are proportional to the quorum size, that is, $f$. Table 1 indicates the minimum and maximum number of messages exchanged in the system according to the fault threshold $f$, for the file storage and retrieval procedures. Comparing the values presented on table 1 with the quorum system size ($3f + 1$ nodes), it can be verified that roughly two messages per node are exchanged in system to store or to retrieve a file. Table 2 presents an estimation of the amount of data exchanged data among nodes in both procedures (best and worst cases). Only messages that transfer files are considered (*storeReq* and *retrieveReply*), as the other messages are much shorter and can be ignored.

**Table 1. Number of messages exchanged.**

| Procedure | best case | worst case |
|---|---|---|
| Storage | $4f + 4$ | $6f + 4$ |
| Retrieval | $4f + 6$ | $6f + 8$ |

**Table 2. Number of file transfers.**

| Procedure | best case | worst case |
|---|---|---|
| Storage | $2f + 1$ | $3f + 1$ |
| Retrieval | 1 | $2f + 1$ |

## 6 Experimental Results

The algorithms presented in section 5 were implemented on *FreePastry*, an open implementation of the *Pastry* environment. The simulation environment was a Pentium 4 CPU 2.60GHz, 1GB RAM running Linux 2.6.19 and SUN JVM 1.6.0_07. File compression used the ZIP algorithm; *TripleDES* was used for file encryption/decryption and MD5 for hashing. All those algorithms were chosen due to their availability in the Java API.

For comparison, we implemented also a simple backup system, in which each node replicates the file in $S$ nodes, where $S$ is the size of the byzantine quorum system. To retrieve a file, the node queries all other nodes for the file, one at a time, until it gets a correct replica. In this simple backup application there is no protocol to ensure intrusion tolerance nor data consistency.

To evaluate our proposal, the number of messages exchanged during the store and retrieve procedures were compared with the simple backup system. Figure 2 presents the number of messages exchanged to store a file in both systems. It can be observed that the quorum system always uses more messages. The results obtained confirm the values in table 1.

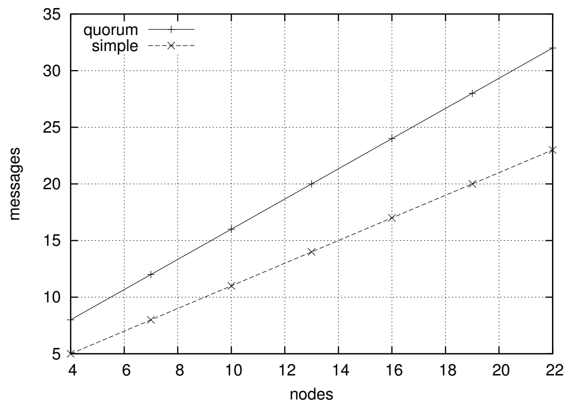

**Figure 2. Messages needed to store a file.**

Figure 3 shows the messages needed to retrieve a file in both systems; the simple backup system has two extreme situations: in the best case, the first replica obtained is correct, while in the worst case only the last replica retrieved is correct. The quorum system fits between these extremes.

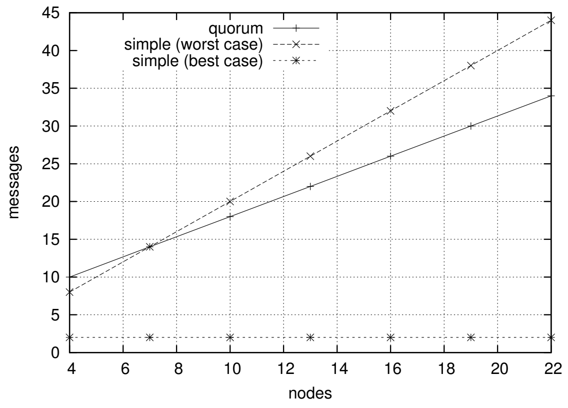

**Figure 3. Messages needed to retrieve a file.**

To analyze the system behavior when facing malicious nodes, a system with 22 nodes was exposed to a variable number of malicious nodes between zero and 7 (as $3f + 1 = 22$). The number of messages needed to store and to retrieve a file was measured. Figure 4 shows that the number of messages varies linearly with the number of malicious nodes.

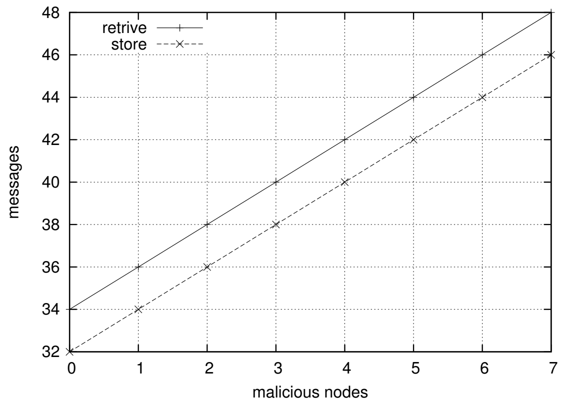

**Figure 4. Impact of malicious nodes.**

## 7 Related Work

The first cooperative backup system, named *CBS*, was proposed in [15]. The CBS has a centralized server to provide the peer location service and uses *erasure coding* along with encryption techniques to improve its fault tolerance. The CBS also deploys some mechanisms to protect itself from attacks, like periodic queries in random blocks and reading limitations. However, this project is more focused on attack prevention than on attack tolerance. *Pastiche* system [5] adopts a probabilistic mechanism to detect malicious nodes through random verification. Pastiche's successor, *Samsara* [6], only protects itself from greedy users, because they are more frequent than malicious users.

The *VentiDHash* system [19] uses erasure coding and cryptographic techniques, like CBS and *PeerStore* [13]; these systems are not directly aimed at increasing attack tolerance. The *pStore* system [2] deals with malicious node faults by replicating and signing data blocks, to prevent that a malicious node impersonates data owners and change or eliminate their data. The *ABS* system [4] mainly focus in the efficient use of local resources, without worrying with intrusion tolerance.

Based in the observation that virus, worms and similar digital plagues may attack computers running a given set of programs, the *Phoenix* system [11] focus in implementing techniques that take advantage of software diversity to provide data backups. It does not focus on intrusion tolerance either, as the *DIBS* system [10], that was proposed for use in local area networks, in which all nodes are considered reliable and trustful.

When compared with the related work, the architecture proposed in this paper has the advantages of being totally decentralized and of adopting a quorum based intrusion and fault tolerance approach, which offers a better level of reliability and lower communication costs.

## 8 Conclusions and Future Work

In this paper we presented a backup system based on byzantine quorum systems, which provides intrusion tolerance, consistency and availability. In addition to the BQS properties, cryptographic techniques were used to provide confidentiality and integrity. The Byzantine Quorum System adopted was based on the *f-dissemination* type, but it was slightly modified to increase efficiency in terms of bandwidth and storage space utilization. The initial results from experimental evaluations show that the number of messages exchanged is proportional to the system size.

The proposed system can be improved in several aspects. First, the usage of an additional register for file hashes at the client side (the node that requested a file to be stored) allows to reduce the BQS size, as the client can use this information to verify the integrity of the received data. Second, as the cooperative backup system uses a *single-writer/single-reader* access semantics, it would be possible to work with quorums of size $f + 1$, reducing the traffic in the network and the storage space used. Finally, a quorum protocol that takes in consideration malicious clients could also be studied, as the system with *single-writer/multi-reader* semantics using the *f-masking* quorum system presented in [7].

## References

[1] A. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J. Martin, and C. Porth. BAR fault tolerance for cooperative services. *SIGOPS Operating Systems Review*, 39(5):45–58, 2005.

[2] C. Batten, K. Barr, A. Saraf, and S. Treptin. pStore: A secure peer-to-peer backup system. Technical Report TM-632, MIT Laboratory for Computer Science, 2002.

[3] J. Cipar, M. Corner, and E. Berger. TFS: A transparent file system for contributory storage. In *USENIX Conference on File and Storage Technologies*, 2007.

[4] J. Cooley, C. Taylor, and A. Peacock. ABS: the apportioned backup system. Technical report, IT Laboratory for Computer Science, 2004.

[5] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. *SIGOPS Operating Systems Review*, 36:285–298, 2002.

[6] L. P. Cox and B. D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *ACM Symposium on Operating Systems Principles*, pages 120–132, 2003.

[7] W. Dantas, A. Bessani, J. Fraga, and M. Correia. Evaluating byzantine quorum systems. In *IEEE Intl Symposium on Reliable Distributed Systems*, pages 253–264, 2007.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[9] A. H. Haeberlen, A. J. Mislove, and P. Druschel. Consistent key mapping in structured overlays. Technical Report TR05-456, Rice CS Department, Houston TX, 2005.

[10] E. Hsu, J. Mellen, and P. Naresh. DIBS: distributed backup for local area networks. Technical report, Parallel & Distributed Operating Systems Group, MIT, 2004.

[11] F. Junqueira, R. Bhagwan, K. Marzullo, S. Savage, and G. M. Voelker. The Phoenix recovery system: rebuilding from the ashes of an Internet catastrophe. In *Ninth Workshop on Hot Topics in Operating Systems*, 2003.

[12] M.-O. Killijian and L. Courtes. A survey of cooperative backup mechanisms. Technical Report 06472, LAAS, Toulouse France, 2006.

[13] M. Landers, H. Zhang, and K.-L. Tan. PeerStore: better performance by relaxing in peer-to-peer backup. In *Intl Conference on Peer-to-Peer Computing*, 2004.

[14] J. Li and F. Dabek. F2F: Reliable storage in open networks. In *Intl Workshop on Peer-to-Peer Systems*, Santa Barbara CA, Feb. 2006.

[15] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *USENIX Annual Technical Conference*, 2003.

[16] B. Liskov and R. Rodrigues. Tolerating byzantine faulty clients in a quorum system. In *IEEE Intl Conference on Distributed Computing Systems*, 2006.

[17] D. Malkhi and M. Reiter. Byzantine quorum systems. In *ACM Symposium on Theory of Computing*, 1997.

[18] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Intl Conference on Distributed Systems Platforms*, pages 329–350, 2001.

[19] E. Sit, J. Cates, and R. Cox. A DHT-based backup system. In *1st IRIS Student Workshop*, 2003.