An experimental evaluation of the Pastis peer-to-peer file system under churn

Fabio Picconi, Jean-Michel Busca, Pierre Sens

▶ To cite this version:

Fabio Picconi, Jean-Michel Busca, Pierre Sens. An experimental evaluation of the Pastis peer-to-peer file system under churn. [Research Report] RR-6114, INRIA. 2007, pp.25. inria-00128869v2

HAL Id: inria-00128869

https://hal.inria.fr/inria-00128869v2

Submitted on 7 Feb 2007

An experimental evaluation of the Pastis peer-to-peer file system under churn

Fabio Picconi — Jean-Michel Busca — Pierre Sens

N° 6114

Février 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6114--FR+ENG

An experimental evaluation of the Pastis peer-to-peer file system under churn

Fabio Picconi , Jean-Michel Busca , Pierre Sens

Thème COM — Systèmes communicants
Projets Regal

Rapport de recherche n° 6114 — Février 2007 — 22 pages

Abstract: This paper presents an experimental evaluation of the effects of churn on Pastis, a DHT-based peer-to-peer file system. We evaluate the behavior of Pastis under churn, and investigate whether it can keep up with changes in the peer-to-peer overlay.

We used a modified version of the PAST DHT to provide better support for mutable data and to improve tolerance to churn. Our replica regeneration protocol distinguishes between mutable blocks and immutable blocks to minimize the probability of data loss. Read-write quorums provide a good compromise to ensure replica consistency under the presence of node failures.

Our experiments use Modelnet to emulate wide-area latencies and the asymmetric bandwidth of ADSL client links. The results show that Pastis preserves data consistency even at high levels of churn.

Key-words: peer-to-peer systems, file systems, churn

Evaluation expérimentale du système de fichiers Pastis dans un réseau dynamique

Résumé : Ce papier présente une évaluation expérimentale du système de fichiers Pastis dans un réseau pair-à-pair dynamique. Nous évaluons le comportement de Pastis afin de déterminer sa capacité à s'adapter aux changements du réseau.

Nous utilisons une version modifiée de la DHT PAST afin d'améliorer la gestion des données modifiables et la tolérance à la volatilité du réseaux, c'est-à-dire, aux connexions et déconnexions des noeuds. Notre protocole de réplication distingue entre les blocs modifiables et immutables, ce qui diminue la probabilité de perte de données. Enfin, un protocole à base de quorums permet d'assurer la cohérence des données modifiables en présence de nœuds fautifs.

Nos expérimentations utilisent Modelnet pour émuler les latences d'un réseau à large échelle ainsi que la bande passante asymétrique des clients connectés par des liens ADSL. Les résultats montrent que Pastis permet de conserver la cohérence des données même à des taux de volatilité élevés.

Mots-clés : systèmes pair-à-pair, systèmes de fichiers, réseaux dynamiques

1 Introduction

DHTs, or Distributed Hash Tables [16, 8, 5], are distributed storage services that use a structured overlay based on key-based routing (KBR) protocols [3]. DHTs provide the system designer with a powerful abstraction for wide-area persistent storage, hiding the complexity of network routing, replication, and fault-tolerance.

One possible application of DHTs is a global-scale peer-to-peer file system. Users contribute some of their free storage capacity, and are in turn allowed to store their files on other nodes of the network. The system stores files persistenly on the large-scale global P2P overlay, allowing users to backup their data, share files with other users, or simply access their own files from any node in the system. The total system capacity can be very high, attracting users who want access to a large number of shared files. Also, since storage is cheap, replication can be used aggressively to achieve high levels of fault tolerance and availability.

In a previous paper we presented Pastis [7], a read-write file system built on top of the PAST DHT. We showed that Pastis achieves acceptable performance even when using a rather strict consistency model such as close-to-open [21], which ensures that applications do not access stale replicas of the file. Although this is beneficial to applications, it also requires keeping track of the existing replicas to enforce the consistency model.

Although replication and consistency have been widely studied since a long time, peer-to-peer systems pose new challenges. One of them is churn, i.e., peers joining and leaving the network rather frequently, which forces the overlay to constantly reorganize itself. Node lifetime observations carried out on unstructured peer-to-peer networks [9, 10, 11, 12] show that it is common for nodes to remain connected to the network only during a small amount of time (e.g., one hour).

Churn can limit the availability of a given object. In order to minimize the cost of object location, most DHTs store replicas at the set of nodes which are closest to the object key, called the replica set [3]. As new nodes join the network, the replica of an object set may change, making some of the replicas unavailable until they are transferred to the new nodes. Conversely, whenever a node leaves the overlay the replicas it stored must be regenerated somewhere else to keep the replication factor constant. Clearly, if the churn rate is too high, the system may not keep up with the network changes, resulting in data being unavailable or lost.

Consistency is also hard to maintain under high churn. Pastis uses replica quorums for read and write operations in order to guarantee consistency in the presence of faulty or Byzantine nodes. A high churn may result in several replicas becoming unavailable, making a quorum impossible and preventing clients from performing further read or write operations.

The original FreePastry implementation of Past did not provide read-write quorums, so Pastis runs on a modified version of the DHT. We also modified Past's maintenance algorithm to distinguish between plain data and quorum-based metadata, as the latter is more sentitive to churn.

We evaluated Pastis under different churn conditions, using Modelnet [15] to emulate wide-area conditions on our local cluster. Our experiments show that Pastis can handle moderate churn levels with little or no data loss.

This paper makes the following contributions. First, it provides an experimental evaluation of our read-write peer-to-peer file system under churn. Second, it shows the need for an enhanced maintenance algorithm for DHTs storing mutable meta-data. Finally, it lists some churn-related issues that still need to be resolved.

The remaining part of this paper is as follows: section 2 briefly describes Pastry, Past, and Pastis. Section 3 presents some shortcomings of the original Past replication algorithm. Section 4 describes our modifications to Past. Section 5 presents our evaluation of the behavior of Pastis under churn. Finally, section 6 presents related work and section 7 concludes this paper.

2 Pastry, Past and Pastis

In this section we present a brief description of the Pastry and Past protocols, and the Pastis peer-to-peer file system.

2.1 Pastry and Past

Pastry [2] is a structured key-based routing (KBR) substrate designed to support a very large number of nodes. In a Pastry network, each node has a unique fixed-length node identifier (nodeid) which is randomly assigned when it joins the network. The nodeid space can be thought of as a circle ranging from 0 to $2^{idlen} - 1$, where $idlen$ is the nodeid length in bits.

Pastry's routing algorithm is derived from the work by Plaxton et al. [1]. The basic idea is the following: both nodeids and the routing key are interpreted as a sequence of base 2^b digits, where b has a typical value of 4. When a message is routed, each hop forwards the message to a node whose nodeid shares a longer prefix with the routing key. If such a node is not known at the current hop, then the next hop is selected so that its nodeid is closer to the routing key. The message is finally delivered to the node whose nodeid is numerically closest to the key. For more information on Pastry see [2, 4].

Each Pastry node maintains a list of its logical neighbors, i.e., those nodes whose ids are numerically closest the local node id. This is called the Leaf Set, and must always contain the $L/2$ closest nodeids clockwise, and $L/2$ counterclockwise. When a node joins or leaves the network, the leaf sets of its logical neighbors are updated so that they always point to the closest node ids in the ring.

Past [5] is a highly-scalable peer-to-peer storage service which provides applications with a distributed hash table (DHT) abstraction. Past uses Pastry to route messages between Past nodes, and in doing so leverages Pastry's properties, i.e., scalability, self-organization, and locality.

Past stores data in blocks, which are replicated for fault-tolerance and performance. The location of a block's replicas is determined by the block's key: a block replicated k times is stored in all the key's i-roots with $0 \leq i < k$, i.e., in the first k nodes whose ids are numerically closest to the block's key. These nodes are called the replica set. Determining the replica set for a given block is straightforward: a message is routed to the node which is numerically closest to the key (the 0-root), which examines its Pastry leaf set to determine which are the next $k-1$ nodes closest to the key.

The main disadvantage of this mechanism is that the replica set of a block may be altered when a new node joins the network. This happens when the joining node becomes the i-root (with $i < k$) for a block's key. At the same time, the $(k-1)$-root will become the k-root, thus leaving the replica set. A new replica should be created on the node that has entered the replica set, and the new k-root should erase the replica from its local store. The replica set for a given block may also change when a node disconnects from the network.

Transferring all the necessary replicas after a node arrival or departure may be quite expensive, and can take a long time. We investigate this in the following sections, and show that this replica transfer mechanism puts a limit to the rate of new arrivals and departures that the DHT can tolerate without losing blocks.

2.2 Pastis

Pastis [7] is a read-write file system that can scale to large numbers of nodes and users. It uses a modified version of Past to store all file system data so that every Pastis client connected to the DHT can access the file system.

As shown in Figure 1, Pastis stores its data in data structures similar to the Unix file system (UFS). The metadata of a file, similar to a UFS inode, is stored in mutable DHT blocks which we call User Certificate Blocks, or UCBs. In order to guarantee data authenticity, each UCB is digitally signed by the writer before it is inserted into the DHT (for more information on UCBs see [7]).

File and directory contents are stored in fixed-size DHT blocks. These blocks are immutable and inserted into the DHT using Content-Hash Blocks, or CHBs. CHB keys are obtained from the hash of their contents, and are therefore self-certifying. The keys of a file's CHBs are stored in the file's inode block pointer table (see Figure 1).

Whenever a file contents are modified, the contents of one or more CHBs change, and will therefore hash to a different value. Thus, writing to a file requires reinserting at least one CHB under a new key, and updateing the inode to point to the new CHB.

CHBs are immutable and therefore do not cause any consistency problems. Inode replicas, however, are mutable and may diverge as the file is updated several times. Pastis implements two consistency models to avoid accessing stale replicas. In the $close-to-open$ model, writes are cached locally until the file is closed, and opening a file ensures that the client will access the latest version available on the network. Converesely, in the $read-your-writes$ model, Pastis only ensures that an open operation accesses a version number that is as high as the previous open issued by the same client. This model is useful for file that are not shared, and achieves better performance as it requires fewer network accesses.

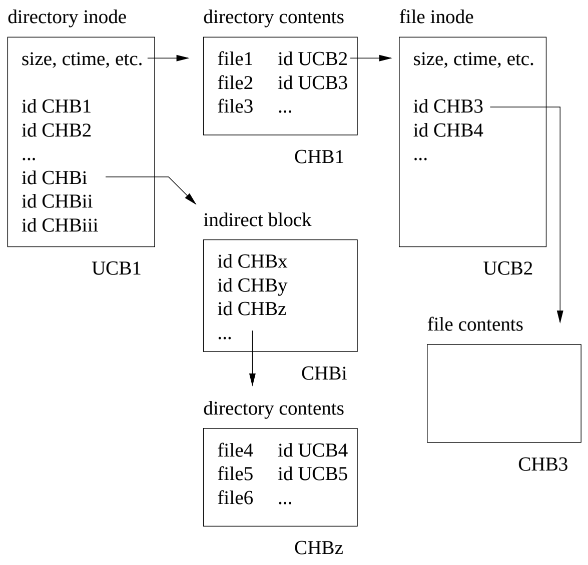

Figure 1: File System Structure

Close-to-open consistency requires reading the latest file version that was written to the DHT, possibly by another client. Inode writes, however, may not update all existing replicas in the DHT. Lost messages or crashed nodes may prevent the update from reaching the entire replica set. Inode read requests may also be directed to nodes that are overloaded or temporarily unreachable. In order to avoid the long timeouts due to these accesses, Pastis uses read-write quorums for DHT operations (see Section 4).

3 Shortcomings in Past's maintenance algorithm

The FreePastry [4] implementation of Past uses a very simple maintenance algorithm that tolerates only small levels of churn.

The algorithm works as follows: node A periodically queries its logical neighbors by sending them a Bloom filter [20] of the keys of the blocks already stored in A. Bloom filters are compact representation of sets of elements that support membership queries. Using A's Bloom filter, a neighbor B determines whether it contains one or more blocks that A should also store but does not possess, and reply to A sending a list of the ids of such blocks. Node A will then add these ids to its list of replicas that must be regenerated, and start fetching them from its neighbors.

This mechanism has several limitations. First, by default a node only queries its neighbors every 10 minutes. This is called the maintenance interval. This means that if several nodes leave the network within less than 10 minutes, nodes may only react after several replicas have been lost.

Second, when a node must regenerate a replica, it will contact its neighbors and transfer the first replica that is found. If the block is mutable, the transferred replica may correspond to a stale version. This means that a single stale replica could potentially be propagated to the rest of the replica set as new nodes join the network (and enter the replica set), eventually polluting the entire replica set.

Third, the original FreePastry algorithm does not implement any fetch priorities when regenerating replicas. As we will see in the following sections, giving a higher priority to the regeneration of mutable block replicas can improve the DHT's resistance to churn.

Finally, the original FreePastry code adds a 500 millisecond delay between each block regeneration. Although this may be useful to avoid congesting the node's Internet connection, it will also lower the block transfer rate. We will show that this may be especially harmful when mutable replicas must be quickly regenerated to avoid data loss.

4 Enhanced Past

Pastis uses a modified version of the FreePastry implementation of Past. It provides better support for storing, accessing, and regenerating mutable blocks. In this section we describe the use of read-write quorums for mutable replicas, and the enhanced maintenance algorithm.

4.1 Handling mutable replicas

As we saw in the section 2.2, the use of replicated mutable blocks introduces the problem of replica inconsistency. Our solution to this problem consists in the use of a standard quorum-based read-write algorithm [19].

The algorithm works as follows: for each modifiable block there is a fix number $k = 3f+1$ of replicas (the replica set), where f is the maximum number of faulty nodes that can be tolerated. By faulty we mean nodes that have either crashed or are acting maliciously. When a client updates a block, it sends the new version to all nodes in the replica set. The operation is complete when a quorum of $2f + 1$ nodes have replied with a positive acknowledgment. When a block must be read back from the DHT, the client queries all the nodes in the replica set, and considers the read operation complete after it receives the responses of $2f + 1$ nodes.

Mutable blocks must be signed by the writer so that nodes in the replica set, as well as clients reading back the replicas, can verify the digital signature to determine if data is authentic. Therefore, we only need the two quorums to intersect in one correct replica to guarantee that data can be read safely.

Note that a malicious node in the replica set cannot forge a signature, but may deny the existence of a replica or return an old version of it (roll-back attack). However, since mutable blocks are version-numbered, a single valid replica in the intersection between write and read quorums is enough to determine the highest version number available in the network.

In our current implementation, a node that leaves the replica set after another node has joined in will erase its local replica. If mutable replicas were kept on such nodes, more than $3f + 1$ replicas could exist at a given time. These additional replicas could return to the replica set when another node leaves the network, thus bringing an old version of the block to the replica set. This version could be read back by a client and be wrongly considered as valid.

Keeping the replicas when a node leaves the replica set would improve the DHT's resistance to simultaneous node joins, but this would require a more complex replication protocol. In our current design we have chosen to trade off churn tolerance (by erasing replicas when nodes leave the replica set) for protocol simplicity.

We should also note the standard quorum protocol ensures atomicity as long as the replica set does not change. However, the replica set does change in practice as nodes join and leave the network. As a result, a situation could arise in which two users have different views of the replica set, and perform read and write operations that may not be atomic. We are currently working on a new algorithm that will preserve atomicity in the presence of changing replica sets.

The quorum-based algorithm provides a compromise between the two primitives provided by the original FreePastry implementation: accessing a single replica (which is risky, since the probability of reading a stale replica is high), and accessing all replicas (which may produce long time-outs when a single replica does not reply).

Finally, we note that this protocol does not apply to immutable blocks, since finding a single valid replica, i.e., whose contents are verified by the hash function, is sufficient for immutable data.

4.2 Enhanced maintenance algorithm

We modified the original maintenance algorithm to avoid the issues we presented in section 3.

First, when a node detects that it must regenerate a mutable replica, it uses the quorum mechanism described in the previous section. This is necessary to avoid regenerating stale replicas, which could potentially lead to losing all up-to-date replicas as new nodes join the network (see section 3). For immutable blocks nodes need only find a single valid replica (i.e., whose contents are verified by the hash function).

Second, modifiable blocks are given higher priority than immutable blocks. The reason for this is that the quorum constraint implies that modifiable blocks can no longer be regenerated after more than f replicas have been lost or erased, as a quorum of $2f + 1$ replicas can no longer be found. We implement a simple priority mechanism in which a node only starts regenerating immutable replicas if there are no pending modifiable replicas left to regenerate.

We assume that applications that use modifiable and immutable blocks will employ a small number of modifiable blocks for storing limited amounts of data (usually meta-data), and a larger number of immutable blocks. Therefore, we expect nodes to finish regenerating

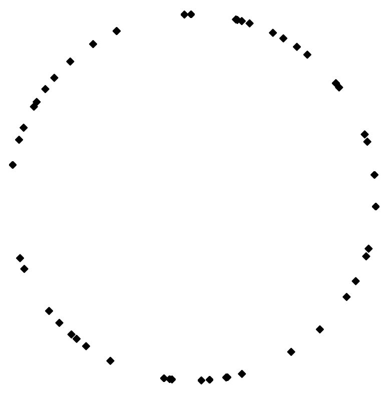

Figure 2: NodeId distribution of the initial 50-node DHT

mutable replicas quickly and start transferring immutable ones before the replica set changes again, and more mutable blocks need to be transferred.

Finally, we eliminated the inter-fetch delay to maximize block regeneration rate. Although this could potentially congest the client's link due to excessive maintenance traffic, our experiments show that this is not so (see section 5.4). In any case, we envisage a block-aware bandwidth-limiting mechanism in which critical modifiable blocks would be transmitted at a high rate, while limiting the bandwidth used for immutable blocks. The design of such a mechanism is left for future work.

5 Evaluation

In this section we present a preliminary evaluation of Pastis under churn. For all our experiments we proceed as follows: first we create a DHT of 50 empty Past nodes, one of which also runs the Pastis client. Although 50 nodes may seem small compared to the expected size of a peer-to-peer network, each join or leave event only affects a few nodes (a block's replica set, which in our case contains 11 nodes).

The Pastis client then creates a new Pastis file system, and copies 400 files of 1 megabyte each, for a total storage of 400 megabytes. Then, new DHT nodes join the ring, which will alter the replica sets of existing blocks. Node arrivals are timed by a Poisson process (except on section 5.6) so that join events are random and bursty. During the experiment, we examine the list of blocks stored by each DHT node in order to determine for each block how many replicas exist at any given time.

In each experiment the initial state of the 50 DHT nodes is the same, namely that of a DHT storing a Pastis file system of 400 1-megabyte files. We use a CHB size of 64 Kbytes, therefore each file is made of 16 different CHBs, plus a UCB for the file's inode, plus one

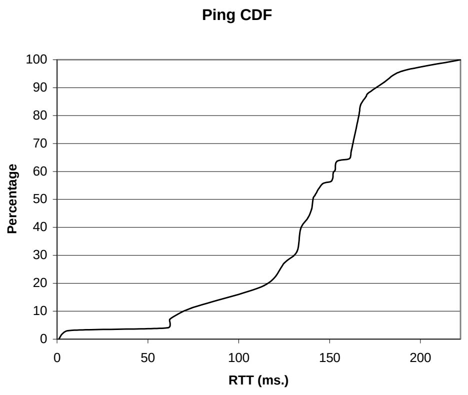

Figure 3: Cumulative distribution function of ping RTTs

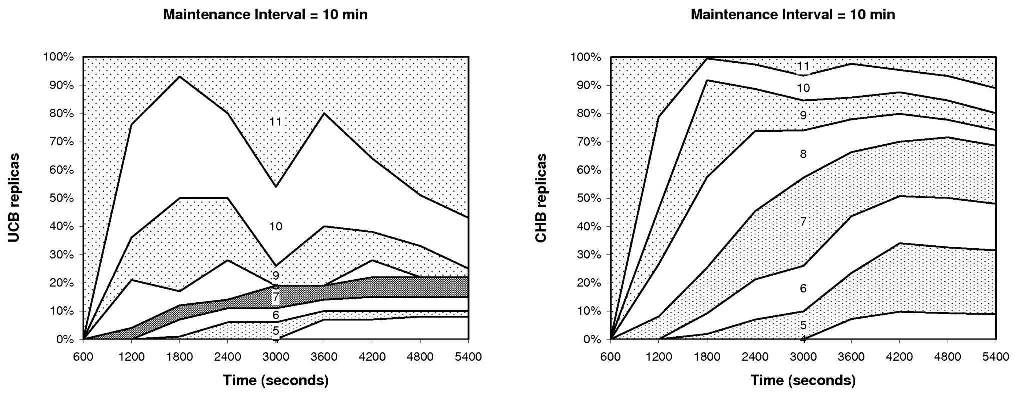

Figure 4: Stacked histogram of number of available block replicas. Node arrival rate: 1 node/min, DHT maintenance interval (Tm): 10 min

CHB for the file's block list. This should yield about 7200 different blocks, although the actual number is slightly larger since each time a file is created the directory's contents are modified, an operation which generates a new CHB.

DHT nodes are configured with a replication factor $k = 11$, i.e., to create 11 copies of each block. This is the maximum number of replicas that can be used using the default 24-node Leaf Sets. With $k = 11$, the system will tolerate up to $f = 3$ failures, resulting in a quorum

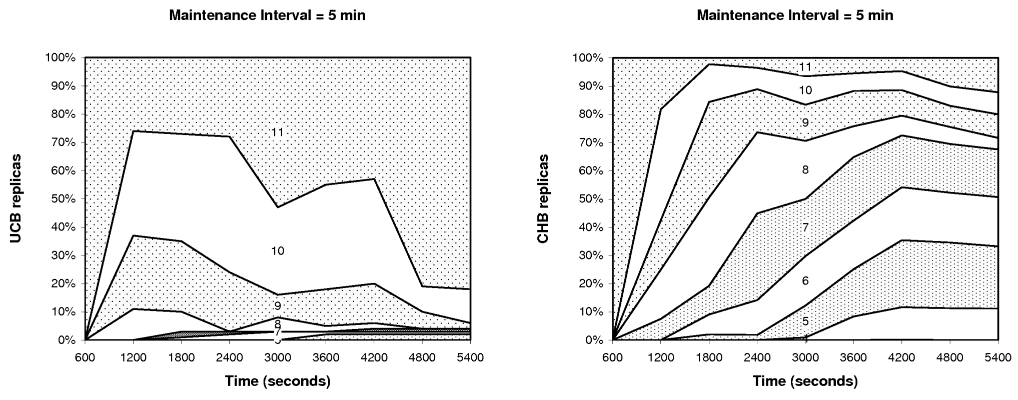

Figure 5: Arrival rate: 1 node/min, Tm: 5 min

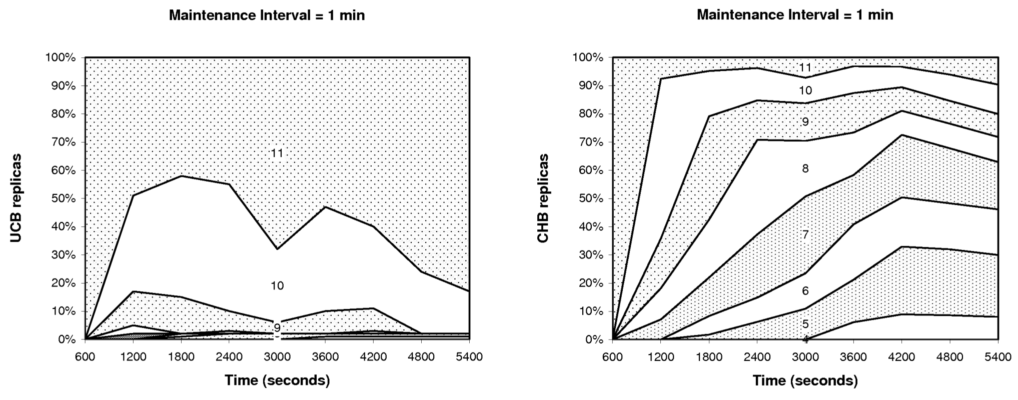

Figure 6: Arrival rate: 1 node/min, Tm: 1 min

size of 8 replicas for read and write operations. This provides the same fault-tolerance as $k = 10$ for mutable blocks, but increased resilience for immutable blocks.

Inserting 7200 different blocks using $k = 11$ generates some 80000 block replicas and 4.4 Gbytes of raw data spread among the initial 50 nodes. This yields an average of 1600 block replicas and 90 megabytes per node, although some nodes contain slightly more and others slightly less. This is due to the fact that nodeids are generated using a hash function on the

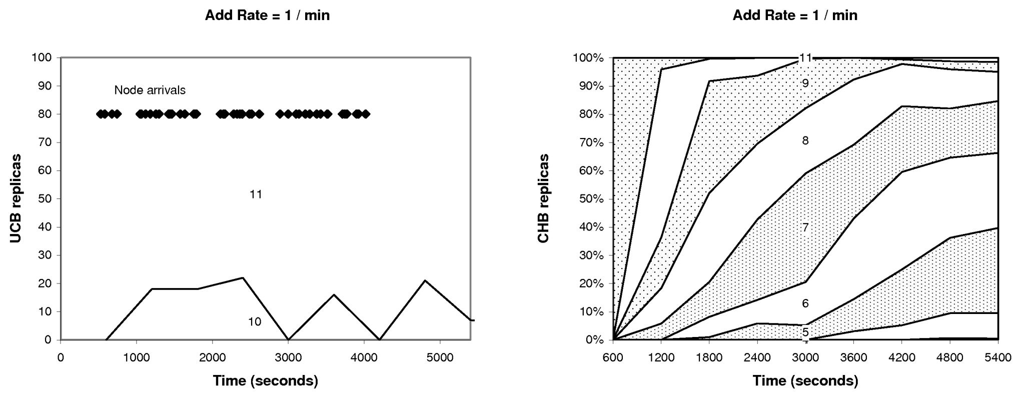

Figure 7: Arrival rate: 1 node/min, fetch delay disabled

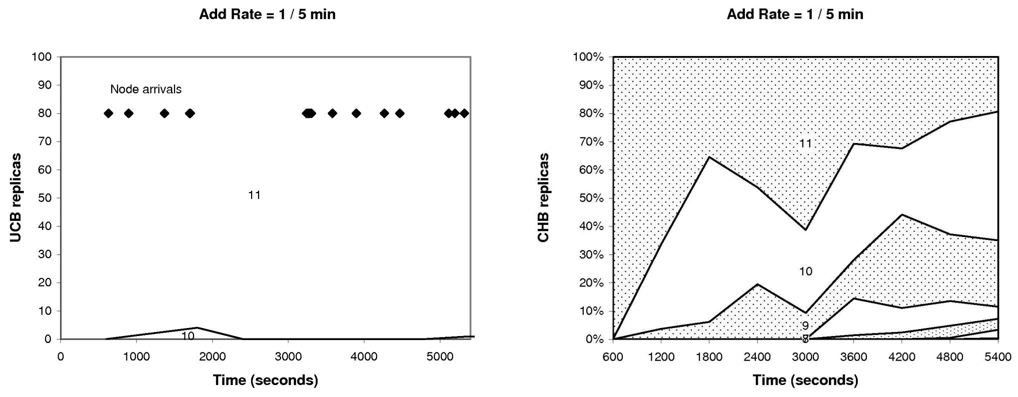

Figure 8: Arrival rate: 0.2 nodes/min, fetch delay disabled

node's IP address, so the resulting nodeid distribution on the ring is not perfectly uniform (see Figure 2).

All nodes run on our local cluster and communicate through a Modelnet [15] emulator core. We run up to 10 DHT nodes per physical host, which are 64-bit dual-Xeon 2.8 GHz with 2 Gbytes of RAM, each running Linux 2.6.x. The Modelnet core is made up of 4 Pentium IV 2.66 Ghz with 512 Mbytes of RAM running FreeBSD 4.11. We use the Inet topology generator along with the scripts provided by Modelnet to create a 5000-node AS-

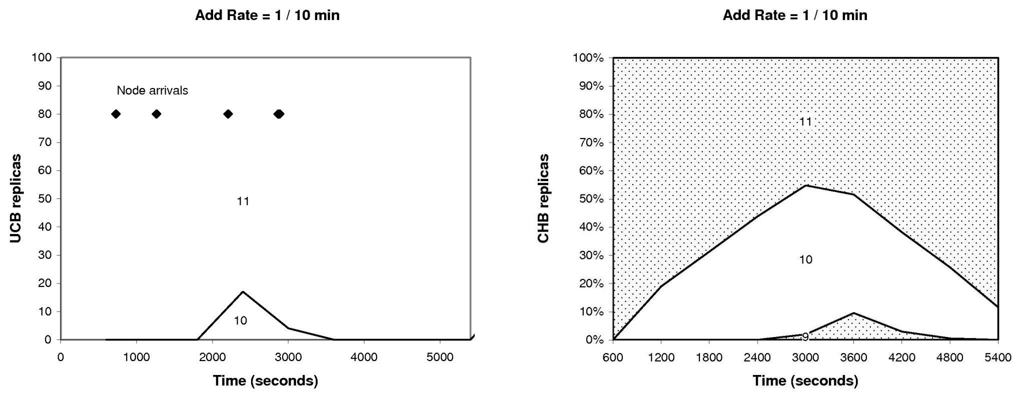

Figure 9: Arrival rate: 0.1 nodes/min, fetch delay disabled

level network with 100 peer-to-peer client nodes connected to 25 different stubs. Link delays are set according to the node location generated by Inet. Figure 3 shows the cumulative distribution function of the ping RTTs obtained by picking one of the nodes and pinging every other peer-to-peer node. For all nodes, the client-stub bandwidth is set to 1 Mbits/s upstream and 10 Mbits/s downstream. This asymmetric bandwidth reflects ADSL2+ link speeds, which is already available in certain countries at the time of this writing. The bandwidth for the other link types (stub-stub, stub-transit, and transit-transit) are set to 10 Gbps, so that only the nodes' ADSL links have a bandwidth-limiting effect.

5.1 Maintenance Interval

In this test we show how the Pastis handles moderate node arrival rates as a function of maintenance interval value Tm. The test starts with our 50 DHT nodes which store the blocks created by the Pastis client after copying the 400 files. We then add 50 new empty nodes according to a Poisson process, using an average rate of 1 node per minute. Since we add 50 nodes, node arrivals stop after approximately 50 minutes.

Figure 4 shows the number of replicas that can be found for modifiable blocks (UCBs, which store Pastis file inodes), as well as for immutable blocks (CHBs, which store file contents). New nodes start joining the overlay at $t = 600$ seconds. The figures show a stacked histogram vs. time of the number of replicas that were found for each block. For each time t, the curve shows the percentage of blocks for which 11 replicas, 10 replicas, etc. can be found across the entire DHT.

The UCB curve shows that at $t = 1800$ the DHT contains at least 10 replicas for 50% of the modifiable blocks. However, for approximately 10% of the blocks only 7 seven replicas or less are available. This is the effect of nodes leaving a block's replica set and thus erasing their local replicas as new nodes join the network. If the maintenance algorithm does not regenerate these replicas quickly enough, more than f replicas may be lost. In this case the block can no longer be regenerated nor be read by a Pastis client, since a quorum is no longer possible.

It is interesting to notice that although any further access to these blocks will fail, there are still several block replicas left in the network. In other words, although the freshness of data can no longer be established (since a read quorum is impossible), data could still be recovered. However, in a real scenario this would require some sort of manual intervention to analyze the data before restoring it.

Figures 4, 5, and 6 show the results for different maintenance intervals. We start with FreePastry's default value of 10 minutes, and then lower it to 5 minutes and 1 minute. The UCB histograms show that the percentage of lost blocks at the end of the test decreases from 20% to about 5% and 2% as we reduce the maintenance interval. This confirms that reacting faster to a lost replica decreases the probability that the number of replicas falls below the quorum threshold.

It is also interesting to notice that the CHB histograms are almost identical for the three maintenance intervals. The reason for this is that there are many more CHBs than UCBs, and that CHBs are also much bigger (64Kbytes versus a few Kbytes). Therefore, CHBs transfer times are much longer than UCBs, and so are CHB fetch queues (which contain the list of blocks that must be regenerated locally). So knowing sooner that a CHB must be regenerated has little impact on overall CHB regeneration.

5.2 Join rate

In this experiment we start with 50 DHT nodes, and add 50 empty nodes according to our Poisson process. The maintenance interval is set to 1 minute.

Figures 7, 8 and 9 show the results for arrival rates of 1, 0.2 and 0.1 nodes per minute. Only the CHB histograms differ significantly. A low arrival rate of 0.1 seems to be handled well by the DHT, since at $t = 5400$ the system has regenerated almost all lost replicas. For the 1 node/minute rate, however, the number of lost replicas seems to increase with time, which would suggest the arrival rate is too high to be handled by the DHT.

5.3 Fetch delay

In this experiment we compare two test runs, one in which we enabled the 500 ms. fetch delay of the original FreePastry code, and the other in which the delay was disabled. These are shown respectively in Figures 6 and 7. Both were obtained using an arrival rate of 1 node per minute.

As expected, disabling the fetch delay greatly improves UCB replication. Our measurements show that the average UCB fetch time is 1200 ms, so introducing a delay of 500 ms clearly slows down UCB replication and therefore increases the probability of losing blocks.

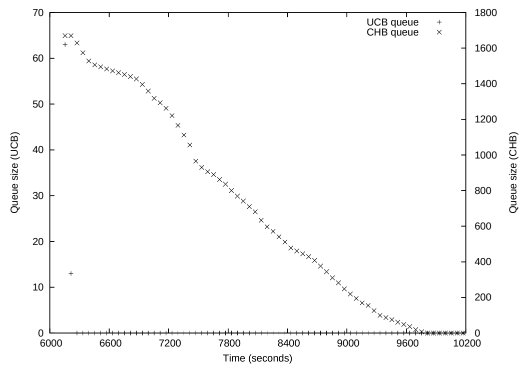

Figure 10: UCB and CHB fetch queue size (arrival rate: 0.1 nodes/minute)

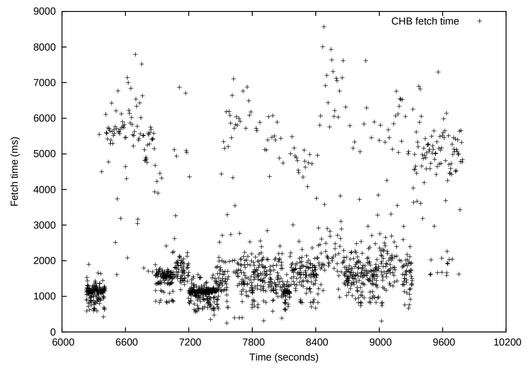

Figure 11: CHB fetch time

5.4 Block fetch queues

Figure 10 shows the size of UCB and CHB fetch queues for a node that has joined the DHT and starts fetching UCBs and CHBs. The arrival rate was 0.1 nodes per minute. The initial sizes are 63 UCBs and about 1650 CHBs. While all UCBs are regenerated in approximately one minute, it takes one hour to regenerate all CHBs. The curve also allows us to estimate the average transfer rate of CHBs and UCBs by dividing the initial queue size by the time is takes to empty it. For CHBs, this yields an average transfer rate of around 2 megabytes per minute, or 30 blocks per minute (the size of our CHBs is 64-Kbytes).

Quite unexpectedly, CHB replication consumes only a fraction of the available client bandwidth. This is probably due to the fact that our immutable blocks are relatively small (64 Kbytes) and that inter-node latencies are high, so overhead is probably very high. CHBs

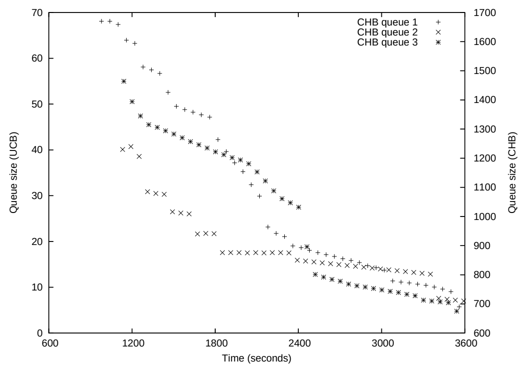

Figure 12: CHB fetch time (arrival rate: 1 node/minute)

fetches could be optimized by requesting several CHBs to be sent on each fetch, thus reducing overhead. This will be left for a future release of Pastis.

Figure 11 shows the CHB fetch times for the same node as Figure 10. We observe that a large number of fetches take around 2 seconds to complete, which confirms the average number of 30 block fetches per minute.

Finally, Figure 12 shows the size of the CHB fetch queues for three of the joining nodes when using an arrival rate of 1 node per minute. The discontinuities in the curves correspond to other nodes (not shown in the figure) joining the network. These new nodes become responsible for a set of the blocks the local node was going to regenerate locally, but no longer needs to since ownership of that replica has moved to the new node.

5.5 Priority to UCBs

As we saw in section 4.2, our modified maintenance algorithm implements a simple priority fetch scheduling: CHBs are only fetched when the UCB fetch queue is empty. To show the need for such a priority-fetch, we performed two test runs, one with the UCB-fetch priority enabled, and the other with the original FreePastry code which does not differentiate between mutable and modifiable blocks. Node arrival rate was set to one node per minute, and the maintenance interval to 10 minutes.

The results are shown in Figure 13. The two curves show the percentage of blocks for which 7 or less replicas have survived, that is, the percentage of UCBs which cannot be read. While fetching UCBs before CHBs leads to a 20% UCB loss rate, disabling the UCB-priority scheduling leads to almost 90% inodes falling below the quorum threshold. This confirms the need to give a higher priority to the regeneration of mutable blocks, which are more sensitive to replica loss than immutable replicas.

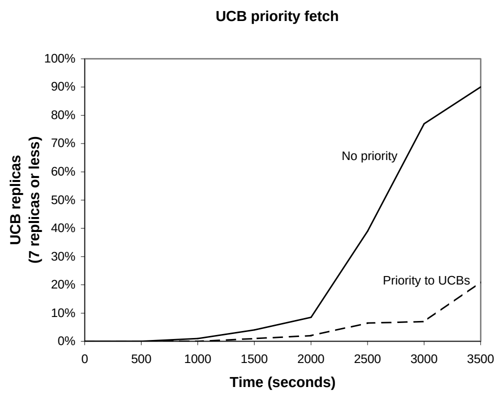

Figure 13: Effect of disabling the UCB-priority fetch

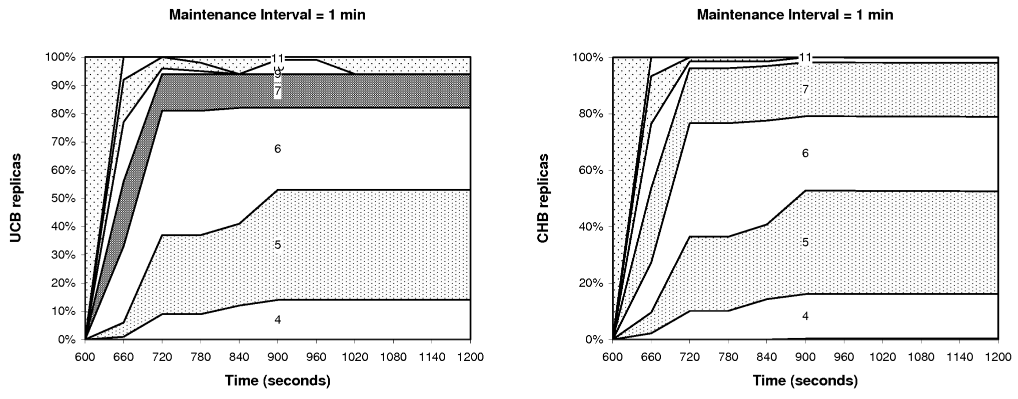

Figure 14: Available replicas after 50 new nodes simultaneously join the DHT

5.6 Massive joins

In this final test we show that the DHT cannot tolerate a large number of nodes join the overlay within a small period of time. The test starts with 50 nodes. We then add 50 new empty nodes, which join the network almost simultaneously. As before, we monitor the number of replicas that can be found for each block.

Figure 14 shows that one minute after the new nodes have begun joining the network, 56% of the inodes have 7 or fewer replicas left in the network. Two minutes after nodes have begun joining only 5% of the inodes have 8 or more replicas. Not surprisingly, the histogram stabilizes with 5% of the inodes showing 11 replicas (missing replicas have been regenerated), whereas the other 95% are simply lost because a quorum cannot be reached.

In practice, a large number of simultaneous joins that affect the same replica sets should be rare, as the ids of joining nodes should be distributed on the entire ring. However, it remains a possible scenario that a robust DHT protocol needs to take into account.

5.7 Modifications to Past

We have shown that reducing the maintenance interval greatly improves replica regeneration and decreases the number of unreadable UCBs. Our modified version employs a 1-minute maintenance interval, compared to the original 10-minute value in Past. Although this increases the bandwidth due to monitoing, this traffic remains small compared to that caused by CHB regeneration. If nodes store 1600 keys each, then every node sends a 800-byte Bloom filter per minute to the other 10 replicas (using 4-bits-per-key Bloom filters), which adds up to 8 Kbytes per node per minute. This is negligible compared to the CHB transfer rate of 2 megabytes per minute we observe.

However, if nodes store a large number of blocks, the size of Bloom filters will also increase. If nodes store 10 Gbytes of data stored in 64-Kbytes blocks, the size of Bloom filters would be 80-Kbytes, and the total bandwidth consumption 800 Kbytes per node per minute. To avoid transmitting such large Bloom filters, nodes could construct a Bloom filter of only the delta of the previously transmitted key set.

Finally, our experiments also show that giving higher priority to UCB regeneration and avoiding the inter-fetch delays of the original Past code also helps improve the effectiveness of the maintenance algorithm.

6 Related work

The effects of churn on structured key-based routing algorithms has been the subject of several recent publications. These studies analyze the characteristics and behavior of different KBR designs under churn, and are obtained either from theoretical analysis, simulations, or experiments carried out on emulated environments.

Liben-Nowell et al. [18] have performed a theoretical analysis of Chord under churn. They determine a lower bound on the maintenance traffic needed to tolerate network churn given the system's half-life, i.e., the time after which half of the node have been replaced by newly arrived ones.

Rhea et al. [13] have shown through emulation that the original Pastry protocol presents routing inconsistencies for a median node session-time of less than a hour. They present and evaluate a new Pastry-based KBR called Bamboo, which routes messages consistently even in the presence of high churn.

Castro et al. [14] present an enhanced churn-resistant version of Pastry called MSPastry. Efficient routing and leaf set maintenance is achieved mainly thanks to active probing and per-hock acks. The authors evaluate their design using simulation.

These studies have focused on tackling the problem of churn at the KBR layer. However, little work exists on the effects of node volatility on the DHT layer, especially regarding replica placement.

Rodrigues et al. [17] have shown that due to bandwidth limitations, using DHTs to store large amounts of data is only viable if node life-times are in the order of several days or weeks. Otherwise the DHT is unable to maintain the replication factor and to guarantee data persistence. However, their work does not study the effects of node sessions (they only consider the case of node definitive departures), nor do they take into account bursty node join and leaves.

7 Conclusion and future work

We have performed an experimental study of the effects of churn on Pastis, a read-write peer-to-peer file system. Pastis' close-to-open model requires the use of read-write quorums to ensure consistency. Quorum-based replication, however, is very sensitive to churn, as disconnected replicas may prevent clients from reaching a quorum.

Our experiments show that Pastis can tolerate moderate levels of churn. We achieve this by using an enhanced DHT maintenance protocol. A priority-based fetch mechanism ensures that meta-data is repaired quickly, while plain data is transferred on the background.

Some problems still need to be addressed. Massive joins may still produce data to become unavailable or lost. This is due to the DHT storing replicas on the nodes closest to the object key. A possible solution would be to store replicas on stable nodes only. The details of such a mechanism are left for future work.

References

[1] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM SPAA. ACM*, June 1997.

[2] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing on large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[3] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *Proc. of IPTPS*, 2003.

[4] FreePastry. http://freepastry.rice.edu

[5] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the ACM Symposium on Operating System Principles (SOSP 2001)*, October 2001.

[6] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation* (OSDI 2002).

[7] J.-M. Busca, F. Picconi, P. Sens : "Pastis: a Higly Scalable Multi-User Peer-to-Peer File System", In *Euro-Par 2005*, Lisboa, Portugal

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (SOSP '01), Chateau Lake Louise, Banff, Canada, Oct. 2001.

[9] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proc. IPTPS*, Feb. 2003.

[10] J. Chu, K. Labonte, and B. N. Levine. Availability and locality measurements of peer-to-peer file systems. In *Proc. of ITCom: Scalability and Traffic Control in IP Networks*, July 2002.

[11] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. ACM SOSP*, Oct. 2003.

[12] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. MMCN*, Jan. 2002.

[13] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Proc. of the 2004 USENIX Technical Conference*, Boston, MA, USA, June 2004

[14] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. Microsoft Technical Report MSR-TR-2003-94.

[15] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proc. OSDI*, Dec. 2002.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[17] R. Rodrigues and C. Blake. When Multi-Hop Peer-to-Peer Routing Matters. In *3rd International Workshop on Peer-to-Peer Systems (IPTPS'04)*, Feb. 2004

[18] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In Proc. ACM PODC, July 2002.

[19] L. Lamport. On interprocess communication. Distributed Computing, pages 77-101, 1986

[20] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of ACM, pages 13(7):422-426, July 1970.

[21] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. ACM Transactions on Computer Systems, 6(1), February 1988.

Contents

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399