Samsara: Honor Among Thieves in Peer-to-Peer Storage

Landon P. Cox, Brian D. Noble
University of Michigan

Presented by:
Todd J. Green
University of Pennsylvania
March 30, 2004
Motivation

A previous system by the same authors was Pastiche, a peer-to-peer, cooperative backup system.

You have some files you want to keep safe by keeping replica backup copies on the net. Pastiche provides a peer-to-peer solution.

Participating nodes offer storage for the files of others, and can store their own files on other nodes.

But Pastiche is vulnerable to abuse, stemming from certain human vices.
Problems with people

- People are malicious. They try to crash systems and wreak havoc. Beyond the scope of this paper.

- People are lazy. They put off preparing presentations until the night before. Beyond the scope of this paper.

- People are greedy. They try to exploit situations to their own advantage. The focus of this paper.
**Greed is Good**

There is incentive to use Pastiche to keep one’s own backups, but little incentive to offer storage for others to use.

Studies of **Napster** and **Gnutella** cited by the authors provide empirical evidence that users tend to consume without contributing in those systems.

The pattern seems to arise frequently. So what can we do about it?
How the Man Wants Us to Do It

The goal is to ensure that nodes consume no more resources than they contribute.

Various solutions have been proposed prior to Samsara, involving

- trusted third parties
- monetary payment
- certified identities

But none of these is peer-to-peer in spirit.
Another Possibility

Another solution that has been proposed to meet the goal is

- symmetric storage relationships

That is, when node $A$ stores a block $a_1$ on node $B$, $B$ stores its own block $b_1$ on $A$.

If $A$ checks and finds that $B$ does not actually have $a_1$ stored, $A$ can penalize $B$ by deleting $b_1$, and vice-versa.

This solves the problem, but the symmetric storage relationship may be unnatural for the application.
Samsara

Samsara is a refinement of the symmetric storage scheme.

Each peer that requests storage of another must agree to hold a claim in return.

Claims are fixed-size blocks of encrypted data (around 4KB).

Peers are held accountable for their claims by the claimholders. The claimholders check on the claims periodically, and may delete blocks stored for delinquent claim keepers.

Claims can be forwarded to “manufacture” symmetric storage relationships.
Claim Structure

A claim is a block of incompressible placeholder data that the host is obliged to keep.

The only subtlety is that the data can be regenerated as needed by the claimholder, from a secret passphrase $P$, a private, symmetric key $K$, and a location in the storage space. Hence no master copy needs to be kept by the claimholder.
Verifying a Claim

Suppose node $A$ has claims $a_1, a_2, \ldots, a_n$ on node $B$, and wants to verify the claims.

We would like to do this using minimal bandwidth (i.e., avoid asking $B$ for the claims in their entirety).

Solution: $A$ sends a key $k$ to $B$. $B$ appends the key to $a_1$ and computes a hash $h_1$; then appends $h_1$ to $a_2$ and computes a hash $h_2$; and so on until it has computed $h_n$, which it returns to $A$. $A$ compares $h_n$ with the value it has computed itself locally.

This is called a query and is performed regularly (e.g. daily) for each replica.
Claim Forwarding

Suppose we have three nodes $A$, $B$, and $C$, and $A$ asks $B$ to store a block $a_1$ while $B$ asks $C$ to store a block $b_1$.

Following the symmetric storage relationship, we have

$$[A : \beta_1] \rightarrow [B : \gamma_1, a_1] \rightarrow [C : b_1]$$

where $\beta_1$ and $\gamma_1$ are claims.

But suppose $B$ wants to reclaim space. $B$ can do this by forwarding $\gamma_1$ to $A$:

$$[A : \gamma_1] \rightarrow [B : a_1] \rightarrow [C : b_1]$$
The Buck Stops Here

Even after a node has forwarded a claim, it still bears responsibility for the claim.

For example, after $B$ forwards $C$’s claim to $A$, if $C$ asks $B$ to verify the claim, but $A$ is down or cheating, $C$ will punish $B$, not $A$.

This is needed to thwart the following simple attack: a dishonest node creates some fictitious identities, and “forwards” every claim to one of its fictitious identities, thereby avoiding responsibility for any claim.
Fragility of Claim Forwarding

Because of the way responsibility is kept even when claims are forwarded, however, a chain of forwarded claims is vulnerable to failure of a single node.

Suppose we have the chain

\[
[A_1 : a_n] \rightarrow [A_2 : a_1] \rightarrow [A_3 : a_2] \rightarrow \cdots \rightarrow [A_n : a_{n-1}]
\]

and suppose that \( A_2 \) fails. What can happen?
Fragility of Claim Forwarding

\[ [A_1 : \alpha_n] \rightarrow [A_2 : a_1] \rightarrow [A_3 : a_2] \rightarrow \cdots \rightarrow [A_n : a_{n-1}] \]

- $A_2$ fails $A_1$’s query for $a_1$
- $A_1$ punishes $A_2$ by deleting (forwarded) claim $\alpha_n$
- $A_{n-1}$ fails $A_n$’s query for $\alpha_n$
- $A_n$ deletes $a_{n-1}$
  
  \[ \vdots \]
- $A_3$ deletes $a_2$
Fragility of Claim Forwarding

So we end up with

\[
[A_1 : \alpha_n] \rightarrow [A_2 : a_1] \rightarrow [A_3 : a_2] \rightarrow \cdots \rightarrow [A_n : a_{n-1}]
\]

leading to

\[
[A_1 : {}] \rightarrow [A_2 : a_1] \rightarrow [A_3 : {}] \rightarrow \cdots \rightarrow [A_n : {}]
\]

All stored data is lost!
Claim Forwarding Cycles

This worst-case situation does not arise if we have a “cycle”. For example,

\[
\leftarrow [A_1 : a_n] \rightarrow [A_2 : a_1] \rightarrow [A_3 : a_2] \rightarrow \cdots \rightarrow [A_n : a_{n-1}]
\]

leads after a failure of \(A_2\) only to

\[
\leftarrow [A_1 : a_n] \rightarrow [A_2 : a_1] \rightarrow [A_3 : a_1] \rightarrow [A_4 : a_3] \rightarrow \cdots \rightarrow [A_n : a_{n-1}]
\]

(Why?)
Claim Forwarding Cycles

“However, if a claim is forwarded back to its creator it forms a dependency cycle, and the claim can be removed from the system entirely.”

\[ \leftarrow [A_1 : a_n] \rightarrow [A_2 : a_1] \rightarrow [A_3 : a_2] \rightarrow [A_4 : a_3] \rightarrow \cdots \rightarrow [A_n : a_{n-1}] \]

In the picture above, $A_1$ and $A_n$ are connected directly, rather than through the claim chain, so the failure of $A_2$ leads only to

\[ \leftarrow [A_1 : a_n] \rightarrow [A_2 : a_1] \rightarrow [A_3 : a_1] \rightarrow [A_4 : a_3] \rightarrow \cdots \rightarrow [A_n : a_{n-1}] \]
Question

$\leftarrow [A_1 : a_n] \rightarrow [A_2 : a_1] \rightarrow [A_3 : \alpha_1] \rightarrow [A_4 : a_3] \rightarrow \cdots \rightarrow [A_n : a_{n-1}]$

Question: If the claim has been removed from the system entirely, then how are subsequent deletions performed?
Fragility of Claim Forwarding

Because of the vulnerability of claim forwarding to node failure, Samsara avoids using it except when absolutely necessary (to reclaim space when a node’s storage is at or near capacity).
Handling Node Failure

Two other questions arise with respect to handling nodes that have failed (or are cheating):

- How exactly is cheating handled? How much data should be discarded when a node is found to be cheating?
- Can we distinguish between transient node failures and cheating?
Handling Node Failure

There needs to be some “grace period” during which nodes can recover from transient failure, and after which nodes are considered to be cheating.

But this can’t be a simple fixed cutoff time, because that permits a simple attack: cheat by moving data to new nodes before cutoff expires.

Samsara solves this and answers the question of how much data to delete with the following solution: punish nodes by deleting their stored data probabilistically and gradually.
Gradual Probabilistic Punishment

Samsara uses a probabilistic model based on the number of failed queries.

For example, after the $i$th consecutive failed query, define the probability that a node will delete a given data block as

$$p_i = \frac{1}{r - i + 1}$$

where $r$ is the normative replication factor (how many replicas tend to exist for a block, on average).
Gradual Probabilistic Punishment

\[ p_i = \frac{1}{r - i + 1} \]

Actually, this is much too harsh, because blocks are small and tend to be very numerous.

A typical Pastiche node will have 2,000,000 objects, equal to about 32GB of state. Letting \( r = 5 \), after the first missed query at each replica site, a Pastiche node can expect to lose 640 objects.
Gradual Probabilistic Punishment

Thus we are motivated to relax the formula to

\[ p_i = \left( \frac{1}{r - i + 1} \right)^{(r - i + 1)} \]

Letting \( r = 5 \) and again with 2,000,000 objects, a Pastiche node can expect to lose an object after missing 3 consecutive queries at all replica sites 7 times.

Curves descend quickly after 3 consecutive queries, however.
Beating Probabilistic Punishment

A dishonest node can still avoid contributing storage by moving data to new replicas before the “expected grace period” expires.

However, this takes too much bandwidth to be practical, according to the authors’ calculations.

(More than ordinary grace period attack?)
Implementation and Experimental Results

They implemented the scheme and ran some benchmarks on it.

Queries run roughly daily.

Chain length short so long as space utilization is relatively low.

Chain length increases pathologically when space utilization approaches 100%.

Reliability with respect to chain forwarding: in their experiments, at 50% utilization, failure of 16% of all nodes results in the loss of only 0.28% of all objects.
Discussion